universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

FactCheck - A Framework for crawling Structured Data

verfasst von | submitted by

Leopold Ferdinand Plonus

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2024

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 935
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Medieninformatik
Betreut von | Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas

Acknowledgements

I want to thank my family, friends, and my girlfriend for their support throughout this research journey.

I am grateful to my supervisor for their guidance and encouragement.

Abstract

Modern web pages often contain semantically rich embedded data to boost their search engine rankings. These data are used by a variety of data consumers to present semantically enriched results to their users. From the widespread adoption of this kind of data also emerges the requirement for the data to be accurate. Currently, data consumers do not receive any support in recognizing and correcting inaccurate structured data on the web. FactCheck++ is a project dedicated to creating methods and tools to recognize and fix conflicting data on the web. In this thesis, a purpose-built focused web crawling system is proposed, with the goal of helping to build a comprehensive knowledge base for the FactCheck++ framework. The crawling system can be configured to focus on topics taken from schema.org, a widely used ontology, computes relevance metrics for visited web pages and is accessible either via a web frontend or a RESTful Application Programming Interface. This work describes the theoretical foundations, design choices, details on the system's implementation, and the experimental evaluation of the prototype.

Kurzfassung

Moderne Webseiten enthalten oft semantisch reiche eingebettete Daten, um ihre Platzierung in Suchmaschinen zu verbessern. Diese Daten werden von verschiedenen Datennutzern verwendet, um ihren Benutzern semantisch angereicherte Ergebnisse zu präsentieren. Aus der weit verbreiteten Nutzung dieser Art von Daten geht auch die Anforderung an die Korrektheit der Daten vor. Derzeit erhalten Datennutzer wenig Unterstützung bei der Erkennung und Korrektur von inkorrekten strukturierten Daten im Web. FactCheck++ ist ein Projekt, das sich der Erstellung von Methoden und Werkzeugen zur Erkennung und Behebung von widersprüchlichen Daten im Web widmet. In dieser Arbeit wird ein fokussiertes Web-Crawling-System vorgestellt, welches das Ziel verfolgt, den Aufbau einer umfassende Wissensbasis für das FactCheck++ Framework zu unterstützen. Das Crawling-System bietet eine Reihe von Konfigurationsmöglichkeiten, berechnet Relevanzmetriken für besuchte Webseiten und ist über ein Web-Frontend und eine RESTful-Anwendungsprogrammier-schnittstelle zugänglich. Diese Arbeit veranschaulicht die theoretischen Grundlagen, Designentscheidungen, Details zur Implementierung des Systems und die experimentelle Evaluierung des Prototyps.

Contents

List of Tables

List of Figures

Listings

1. Introduction

Modern websites often employ structured data markup to boost their search engine rankings. The most widely used format for these data is assumed to be Schema.org with over 12 million sites [1]. Schema.org was introduced in 2011 by Bing[1], Google[2], and Yahoo[3], with the goal to provide a single schema and vocabulary covering a wide range of topics, which should replace the range of schemas that existed and competed at the time. Not only search engines use these data to enrich their results with e.g., info boxes. Other data consumers like knowledge engines (e.g., WolframAlpha[4]) or virtual assistants (like *Apple Siri* or *Amazon Alexa*) use knowledge graphs that are built upon that data. Relying on the quality and correctness of structured data on the web can lead data consumers into presenting false information to end users. This happened during the 2016 Austrian presidential election when Google Search falsely presented Norbert Hofer as the new Austrian president in an info box [2].

FactCheck++ is a project of the Multimedia Information Systems research group, which has the goal of helping data consumers and providers to identify and fix conflicting data on the web. To achieve this goal, vast amounts of data must be collected, categorized, and compared, which is a non-trivial task given the size and complex structure of the World Wide Web (WWW). To streamline this process, a tool is needed that is able to discover sources on the web that are likely to contain relevant data on topics that are of interest for the FactCheck++ project. This thesis covers the theoretical foundation, design, and implementation of such a tool, a focused web crawling system.

1.1. Motivation

With the exponential growth of the WWW, data availability has vastly increased. The WWW is estimated to consist of over four billion web pages [3], containing information on virtually every topic and entity. However, the heterogeneous nature of this information, coupled with its volume, makes it challenging for users to find data relevant to their specific needs, and even more challenging to assess the quality and correctness of the data, as well as the credibility of the data providers. The FactCheck++ project aims to assist users and data providers on the WWW in getting aware of conflicting data. To be able to fulfill this purpose, it is essential to gather vast amounts of data about a large variety of topics. The current state of FactCheck++ relies on manual selection of resources for a limited range of entity types – persons, movies, books, products, and events. As the manual selection and discovery of new web pages and domains to process is a time-consuming task, it prevents fast scaling of the FactCheck++ framework. A (semi-) automated system with the purpose of discovering sources with relevant information about topics of interest could assist in scaling the framework and collecting facts from the WWW more quickly.

The most obvious approach to find relevant web pages for a given topic would be to use a search engine like Google or Bing, as these services provide results fast and use well

[1] https://www.bing.com
[2] https://www.google.com
[3] https://yahoo.com
[4] https://www.wolframalpha.com

established relevance and ranking algorithms. Certainly, this method can be a starting point for the proposed system, but solely relying on the results of established search engines does not suffice, for the following reasons:

By treating one search engine as a single source of truth, the diversity of the collected data is limited, as modern search engines customize search results based on user behavior, location, device [4]. Furthermore, search engine results are often complemented by advertisements and sponsored content which can also undermine the neutrality of search results.

Search engines also provide limited customization and configuration possibilities. While established search engines like Google allow the use of advanced query operators that can significantly improve the precision of results [5], these operators are not well adopted amongst users and are not specifically designed for the use case of FactCheck++. For example, it is not possible to explicitly filter for an entity type; a Google search for "Steve Jobs" returns results for the person Steve Jobs, as well as for the movie that is named after the person. Processing and evaluating these search results would again require manual inspection and filtering. Furthermore, the methods of relevance assessment and ranking of web pages that search engines use are not publicly available, and search engines do not provide information on the degree of relevance of their results beside their position in the result set. This makes it hard to comprehend which features of a web page are used to determine relevance, and at which position in the result set returned pages start to become irrelevant.

A focused crawling system that features an appropriate level of configurability and transparency could be advantageous for the particular use case of FactCheck++. Another disadvantage of search engines is that their index is updated in – sometimes irregular – intervals, which means that results may be outdated [6]. A system that crawls and processes web pages in real-time avoids this problem and guarantees freshness of the provided data.

This is why a general web search engine cannot replace a purpose-built solution, but could support it, by cherry-picking the desired functionalities of a search engine and building on top of those. As Lee et al. showed, augmenting a focused web crawler with the capabilities of a search engine like Google can reduce implementation effort while achieving satisfactory performance [7].

1.2. Goal

The goal of this master's thesis project is to develop a focused web crawling system that supports the FactCheck++ framework in the discovery of web resources that contain relevant information about entities. To accelerate the growth of the FactCheck knowledge base (called FactBase), a service that automatically seeks out new domains and websites which contain relevant information about an extended spectrum of entity types is needed. The proposed system should be able to present results in real-time; this is necessary to satisfy immediate information needs of end users interacting with FactCheck's client application (a browser plugin called IdaFix) and allows asynchronous processing of results.

To ensure that the system effectively differentiates between relevant and irrelevant web pages, the design and implementation of relevance metrics are crucial. These metrics should reflect the system's confidence that a given page contains relevant information on

the queried entity or entity type, such that the data produced by the crawling system can be interpreted by users and other FactCheck++ services that might use those data to extract relevant information from web pages.

1.3. Outline

The thesis is structured in the following way:

Chapter 1 introduces the general topic of this work, the motivation for the project, and its goals.

Chapter 2 serves as the introduction to the field of research and the background of this project. It contains an introduction to the nature of data on the WWW, background information on the FactCheck++ project, and an introduction to the field of web crawling in general.

Chapter 3 breaks down the design of the proposed prototype and considerations that have been made in the design phase of this project.

Chapter 4 contains details of the proposed system's implementation and insights into the used technologies and methods.

Chapter 5 explains the methods used to evaluate the prototype and the results of the evaluation are presented.

Chapter 6 discusses design decisions and possible limitations of the proposed system.

Chapter 7 provides insights into further development possibilities for this project and ideas that could improve the system's performance and cost efficiency.

Chapter 8 summarizes the previous chapters and concludes the thesis.

2. Background & Related Work

This chapter introduces the essential topics that this project is related to, provides insights on related work, and establishes the context within which the focused web crawling system was developed. These foundational concepts are crucial for the development of the proposed system.

2.1. Data on the Web

The WWW was developed by Tim Berners-Lee and his colleagues at CERN during the early 1990's. As Berners-Lee et al. stated in a paper presenting the WWW, it was developed to be a pool of human knowledge [8]. Essentially, the WWW is a collection of documents, encoded in Hypertext Markup Language (HTML), and uniquely identified by a Uniform Resource Identifier (URI). These documents can be interchanged with the use of the Hypertext Transfer Protocol (HTTP).

The web quickly developed into a widely adopted tool that was used by humans to exchange information. Berners-Lee et al. began to envision an extension of the WWW, which should enable non-human consumers of the web to make use of the vast amount of information; the Semantic Web [9]. This goal should be achieved by employing XML (eXtensible Markup Language) and RDF (Resource Description Framework) technologies to embed machine-readable, standardized metadata into web pages. During the development of the Semantic Web, many new approaches and technologies emerged, that should improve the Semantic Web and simplify contributing. Ontologies were developed to reduce ambiguity of different vocabularies being used to describe a web page's content (e.g., a person's date of birth could be described by the attribute "date-of-birth" on one web page, and by "birthday" on another page). But still, the large number of different vocabularies, technologies, and standards in the realm of the Semantic Web was confusing for webmasters, resulting in little participation.

To unify the different available vocabularies and to establish a standard that promotes wide adoption, Google, Bing and Yahoo collaborated and introduced Schema.org in 2011. Schema.org defines a set of entity types and properties, that can be used to describe a wide range of things or concepts independent of domain. The design of schema.org is based on inheritance, which means, that each entity type inherits the properties of its parent. The origin for all other entity types is the type "Thing". Schema.org is now assumed to be the most widely adopted vocabulary for structured data [1].

To include structured data in a web page, multiple formats exist. JSON-LD (JavaScript Object Notation for Linked Data) could be assumed to be the most widely adopted format, as it is recommended by Google [10]. JSON-LD is based on JavaScript Object Notation (JSON) and is usually included in the header of web pages using a <script type="application/ld+json"> tag.

```
<script type="application/ld+json">
    {"@context": "https://schema.org", "@type": "Person",
       "givenName":  "Angela",  "familyName":  "Merkel",  "birthPlace":
    "Hamburg"}
</script>
```

Listing 2.1.: Example of structured data in JSON-LD format

Another structured data format is Microdata, which is part of the HTML living standard [11]. Structured data in Microdata format is usually spread over a HTML document, where each HTML tag can be enriched by specific Microdata attributes that indicate the kind of information that is contained in the HTML tag.

```
<div itemscope itemtype="http://schema.org/Person">
    <p>
        <span itemprop="givenName">Angela</span>
        <span itemprop="familyName">Merkel</span>
        was born in <span itemprop="birthPlace">Hamburg</span>.
    </p>
</div>
```

Listing 2.2.: Example of structured data in Microdata format

The third widely adopted structured data format is RDFa (Resource Description Framework in Attributes), which has a very similar syntax to Microdata, where each HTML tag can be enriched by specific attributes. RDFa is a HTML5 extension and a W3C recommendation [12].

```
<div vocab="http://schema.org/" typeof="Person">
    <p>
        <span property="givenName">Angela</span>
        <span property="familyName">Merkel</span>
        was born in <span property="birthPlace">Hamburg</span>.
    </p>
</div>
```

Listing 2.3.: Example of structured data in Microdata format

While structured data enables machines to interpret web page content in straight-forward manner, most of the content on the web is still designed to be consumed by humans. This type of data can occur in unstructured or semi-structured format.

As the name suggests, unstructured data does not follow any standardized schema and can be in the form of free text, or multimedia like images, audio, or video. Unstructured data requires more advanced methods to be interpreted, extracted, or parsed by machines, including techniques from the fields of Natural Language Processing (NLP) and Machine Learning (ML). Semi-structured data also does not follow a standardized vocabulary or schema but is presented in a somewhat ordered manner. For example, HTML tables and lists could be considered semi-structured data.

2.2. Fact-Checking

As Zhou and Zafarani [13] state in their Survey of Fake News, “The explosive growth in fake news and its erosion to democracy, justice, and public trust has increased the demand for fake news detection and intervention”. Manual fact-checking is a non-trivial and resource-intensive task [14], [15], that struggles to cope with the vast amount of misinformation generated [13]. News organizations and specialized fact-checking services

like Snopes[1] or PolitiFact[2] perform expert-based manual fact-checking. The conclusions drawn by these organizations and services are recognized for their high accuracy [13].

An alternative manual fact-checking approach employs crowdsourcing, where a substantial number of individuals congregated on platforms like Amazon Mechanical Turk[3] perform those tasks on a larger scale. Such strategies could be considered more effective as they leverage the collective ability of the masses to not only detect inaccuracies but also recognize biased information [16]. However, crowd-based methods cannot guarantee a sufficient level of credibility, accuracy, and scalability [13].

The scalability issue is addressed by automated fact-checking strategies that employ techniques like Natural Language Processing (NLP), Information Retrieval (IR), Information Extraction (IE) and Machine Learning (ML) [14]. These automated approaches often still depend on expert-based manual fact-checking and, as Tschiatschek et al. [17] state, should generally augment expert validation, rather than trying to replace it.

2.3. The FactCheck++ Project

FactCheck++ is a project of the Multimedia Information Systems research group with a focus on identifying and fixing conflicting data on the web. It implements a framework that enables users to "become aware of conflicting structured data so that she can recognize a potential problem, confirm correctness or defectiveness of structured data and give feedback to data owners, analytic software, Web services, or crawler, in order to confirm correctness or to provide accurate data for the rectification of defective structured data [...]" [18].

The approach of FactCheck++ is based on the idea, that modern web pages often contain structured information in the form of semantically rich embedded data. These data are provided in a variety of semantic markup formats like Microdata, RDFa, Microformats, and JSON-LD, which are designed to be processed by machines – mostly by search engines to provide semantically enriched search results. This property allows structured data to be compared between web pages in an automated way. FactCheck++ breaks down structured data on the web into two categories:

- **Entity –** A self-contained thing or idea, either fictional or non-fictional. For example, people, movies, or locations.
- **Fact –** A piece of information that describes an entity, represented as key-value pair. For example, the name of a person, or the release date of a movie.

The FactCheck++ framework is a constantly growing software project that incorporates various components and services that are capable of extracting, processing, storing, and presenting data – or facts. The most important components of the FactCheck++ infrastructure are listed below:

[1] https://www.snopes.com
[2] https://www.politifact.com
[3] https://www.mturk.com

- **FactServer –** This is the core component of FactCheck++. It exposes an API for retrieving data about specific entities and facts and enables comparison of these instances. FactServer is also responsible to keep the FactBase updated, in the case of new entities or facts being added, or existing ones are changed.
- **FactBase –** This is the core database of the framework. It stores all facts and entities that are collected by the different FactCheck++ services.
- **FeedbackBase –** This component stores user feedback about potentially conflicting data on web pages. This information is used to refine the precision metrics of the system.
- **Precision Metric Manager –** This component uses unsupervised learning to either establish new or adapt existing metrics, based on feedback from the FeedbackBase.
- **IdaFix Plugin –** IdaFix, named after its function to "identify and fix structured data on the web", acts as the framework's frontend. It extracts structured data from websites, which is then sent to the FactServer for processing. After processing, IdaFix shows the results to the user. It also helps keep the FactBase updated by seeking comparisons for new or revised data, employing the FactServer API. Users can also provide feedback through IdaFix.

Extracting facts and entities from given web pages is implemented for many of the types of data that occur on the web, including embedded structured data, semi-structured data in the form of HTML tables and lists, and for unstructured data like free text or images. A major milestone of the FactCheck++ project is the discovery of web resources that contain relevant information that is worth being extracted into FactBase. This work focuses on the implementation of a tool that can help reaching this milestone.

2.4. Web Crawling

Web crawling, or web scraping, refers to an automated method used to extract large amounts of data from websites. The basic naive web crawling strategy as described in [19] is a simple algorithm, based on the principle of breadth-first search; given one or more seed Uniform Resource Locators (URLs), visit the pages addressed by the URLs, process their content, extract hyperlinks and iteratively visit the pages addressed by those hyperlinks.

Web crawlers are used for a wide range of different purposes, like data mining, web archiving, or website maintenance. The most prominent use case for web crawlers is web indexing. Search engines crawl web pages and index them, to allow users to query the index for pages that match the query.

The first web crawler is assumed to be the World Wide Web Wanderer – also known as The Wanderer, which was developed by Matthew Gray at MIT in 1993. Its purpose was to measure the size of the WWW, which it did until 1995. The Wanderer also helped creating the first web index, the Wandex.

The basic web crawling approach comes with inherent challenges, some of which are also listed in [19] and which became evident during the development of FactCheck++:

1. **Scale –** Broad coverage can only be achieved by crawlers with extremely high throughput which in turn imposes difficult engineering challenges. Tackling this issue by providing the system with excessive amounts of computing resources is an approach used by large corporations with an accordingly large financial budget.
2. **Content selection tradeoffs –** Crawling the whole web is not feasible, even for the highest-throughput crawlers. Instead, crawling should be performed selectively, to collect relevant content quickly and bypass irrelevant content. Without parametrization and logic to restrict the following of hyperlinks, a crawler cannot achieve this goal.
3. **Social Obligations –** Unrestricted crawlers can have an impact on the performance of a visited site by overloading its servers. Ideally, crawlers should implement a politeness policy to prevent unintended denial-of-service attacks.
4. **Adversaries –** Some web sites contain useless or misleading content or hyperlinks that serve the purpose of directing traffic to commercial websites. This behavior is mostly motivated by financial interests and increases the amount of "dead ends" for the crawler.

Considering the challenges presented by basic web crawling, focused crawling, as introduced by Chakrabarti et al. [20], emerges as a viable alternative for obtaining more relevant information in a shorter time. Focused crawling aims to collect data from the web that is specifically related to a predefined set of topics or criteria, rather than indiscriminately scraping data from all possible web pages. In focused crawling, the process begins with one or more seed URLs, similar to a general-purpose crawler. However, instead of following all outbound links, the focused crawler evaluates the relevance of each linked page based on predetermined criteria. This could involve techniques such as keyword density checks, or semantic analysis.

Focused crawling helps solving some of the problems of classic web crawling with the following advantages:

- **Resource Efficiency –** Focused crawlers use computational resources more efficiently, as they only visit pages that are likely to contain information of interest.
- **Relevance –** The dataset obtained through focused crawling is more cohesive and relevant to the specific subject matter.
- **Reduced Impact –** By avoiding unnecessary visits to irrelevant pages, focused crawlers are less likely to overload servers, thus better adhering to social obligations.

2.5. Web Mining

Web mining refers to the process of extracting valuable information from web data sources and is closely related to web crawling. These data sources can include web pages, server logs, and databases. The goal is to discover patterns, extract useful information, or gain insights that can be used in various applications like recommendation systems, search engines, and data analytics. Web mining can be broadly classified into three categories as explained in [21]:

- **Web Content Mining** has the goal of analyzing the content of web pages, including text, images, and other media, to derive useful insights.
- **Web Structure Mining** focuses on the structure of hyperlinks within the web itself. Links between pages can represent relationships that can be analyzed to understand the architecture or topology of a particular website or set of websites.
- **Web Usage Mining** involves mining the user interaction and behavior data, often stored in web logs, to enhance user experience, or to create adaptive websites that change in response to user behavior.

Various techniques are applied in web mining, such as:

- **Text Mining –** To analyze the text found in web pages.
- **Clustering –** To group similar web pages or users together.
- **Classification –** To categorize web content or predict user behavior.
- **Association Rule Mining –** To discover interesting relationships between different web pages or between user behaviors.

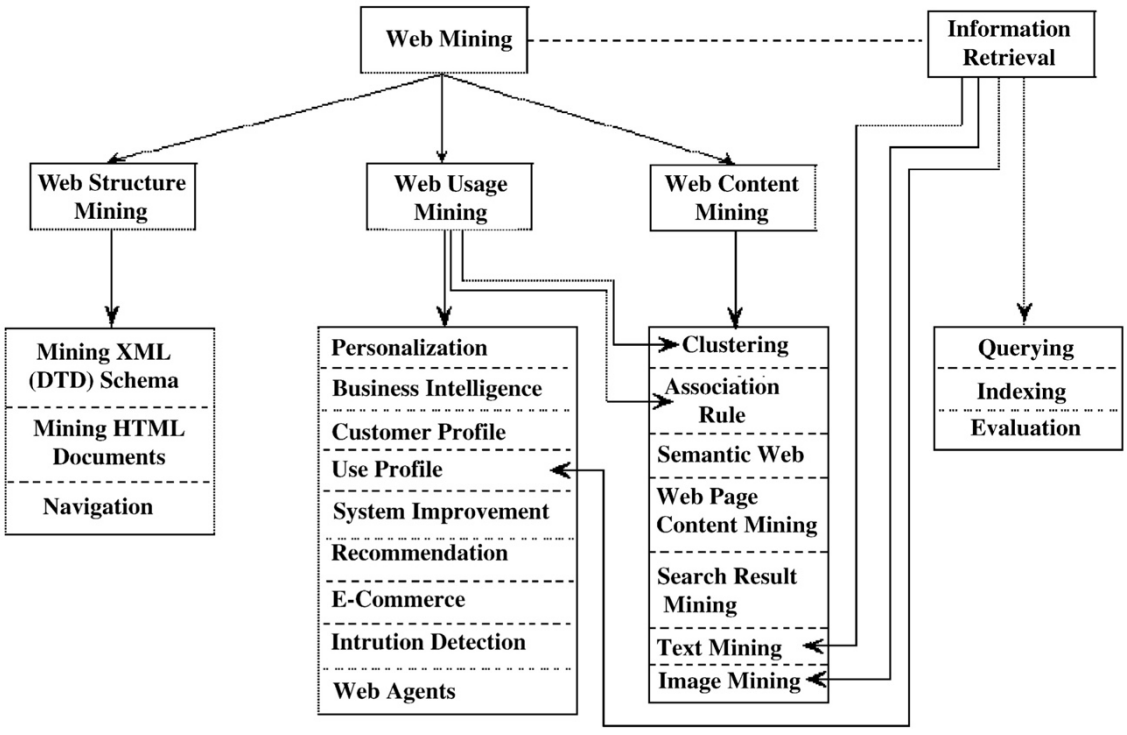


Figure 2.1.: Web Mining Taxonomy as defined by Arotaritei and Mitra [21]

Focused crawling systems, like the one developed for this thesis, employ web mining to achieve higher rates of relevant pages that are crawled. By analyzing the content of web pages and links between pages, the system evaluates the relevance of each crawled page and the links that are found on the page.

3. Foundation of Conceptual Design

This chapter covers the foundations of the conceptual design of the proposed focused web crawling system, as well as requirements that must be met by its implementation, and considerations that influenced the design choices. The conceptual elements presented in this chapter serve as the foundation of the prototypical implementation covered in chapter 4. However, some design choices changed during the implementation phase of the project, to react to unforeseen technical challenges.

3.1. Requirements

This section outlines the crucial functional and non-functional requirements that the focused web crawling system must fulfill to achieve the objectives of this project.

3.1.1. Functional Requirements

1. **URL Frontier Management –** The system must manage a list of URLs to be visited. It should ensure that no URL is visited twice, and that relevant URLs are prioritized.
2. **Acquisition of Seeds –** The system must be able to identify relevant seed URLs for a given query.
3. **User Interface –** The system must provide a user interface that allows users to configure and trigger crawling sessions, monitor the status of sessions, terminate sessions, and investigate the results of crawling sessions.
4. **Web Page Fetching –** The system must be capable of sending HTTP requests to the URLs from the frontier and handle HTTP responses, including various status codes.
5. **Content Parsing –** The system must parse the received HTML documents and extract useful information such as URLs.
6. **Configurability –** The system must allow users to configure its operation through input parameters.
7. **Relevance Determination –** The system should assess the relevance of a web page, or a URL based on predefined criteria. The degree of relevance should be denoted by normalized, numerical values.
8. **Data Persistence –** The system must be able to persist crawling results and session data.

3.1.2. Non-functional Requirements

1. **Efficiency –** The crawler should make efficient use of network, computational, and storage resources to ensure that the crawling process is as quick and cost-effective as possible.
2. **Robustness –** The system should be capable of handling errors gracefully. It should also be able to recover and resume operation after failures.
3. **Scalability –** The system should be designed in a way that allows it to scale up to handle larger volumes of data as needed.
4. **Respect for Web Standards and Ethics –** The crawler must adhere to web standards such as the Robots Exclusion Protocol. It should also respect website terms and conditions and avoid causing disruption to the services of the websites being crawled.
5. **Usability –** The systems user interface should be intuitive and easy to use. It should allow users to easily configure the crawler and monitor its progress.

3.2. Ontology Selection

An ontology is a formal representation of knowledge as a set of concepts within a domain and the relationships between them. In the context of the Web, ontologies provide a shared and common understanding of a domain that can be communicated across people and applications. Ontologies are extensively used in artificial intelligence, semantic web, software engineering, and information architecture as a form of knowledge representation about the world or some part of it.

The crawling system should have knowledge about a broad spectrum of entity types and their properties, to be able to decide whether content on the web references an entity of a specific type. The schema.org ontology is a good fit for this use case, as it is the de facto standard for structured data on the web and has a rich vocabulary that caters to a large number of entity types.

To facilitate this ontology not only for dealing with structured data, but also for unstructured data, we decided to extend the schema.org type names and property names by synonyms. For example, in an unstructured context, the schema.org type “Bakery” could also be referenced by “pastry shop”, “patisserie”, or “cake shop”. The schema.org property “birthDate” could be referenced by “birthday”, “date of birth”, or “born”. By equipping every type and property with synonyms, we allow the crawling system to recognize occurrences of types and properties with higher precision.

The goal of recognizing entities and their properties in free text could also be achieved by other commonly used techniques in the field of Natural Language Processing (NLP). The state of the art for Named Entity Recognition (NER) and Information Extraction (IE) includes the use of neural networks or transformer models like BERT (Bidirectional Encoder Representations from Transformers). Those techniques produced promising results for a multitude of applications, but only for domain specific use cases [22], [23].

In detail, the rule-based approach was chosen because of the following benefits:

- **Transparency –** Rule-based systems are generally easier to understand and interpret. Decisions of the systems can be retraced, which is important for debugging and development.
- **No Annotated Data Needed –** Rule-based systems don't require large sets of annotated data for training, which can be time-consuming and expensive to produce.
- **Predictability –** Rule-based systems are deterministic in that they make decisions based on a predefined set of rules, making their behavior more predictable compared to machine learning models.
- **Domain Independence –** Machine Learning techniques have shown to perform well in domain specific use cases. By designing rules based on a domain agnostic schema like schema.org, the system could perform well regardless of the domain or topic that is of interest to the user.
- **Multilingual Support –** The system should provide reasonable extension possibilities regarding supported languages. With the chosen approach, it is relatively easy to support other languages than English. A machine learning approach would require training data in each new language.

3.3. Input Parameters

To guide the focused crawling process effectively and to ensure that the most relevant data is retrieved, the crawling system can be configured using input parameters. These parameters allow users to tailor the crawl according to their specific requirements. The following parameters have been chosen to provide a reasonable level of configurability for each session:

1. **Entity Name** – The name of the desired entity.
2. **Entity Type** – The schema.org type of the desired entity. Together with the name of an entity, it helps to pinpoint the exact entity the user is interested in.
3. **Minimum Score** – The minimum relevance score that must be met to persist and return the result. This can help avoiding irrelevant content to be stored in the crawlers database. Furthermore, it is unlikely that users are interested in receiving results with low relevance scores, so this parameter allows them to precisely dial in the degree of relevance that they are willing to accept.
4. **Time Window** – The maximum amount of time the crawler should run. This gives users more control over the crawling process, and allows for a “quick scan” that could be beneficial in cases where IdaFix users submit an ad-hoc request for a unknown entity or type and expect fast results.
5. **Seeds** – URLs that can guide the crawler to other relevant web pages. Appropriate seed selection can improve crawling efficiency and effectiveness [24]. While the crawling system should be able to acquire high quality seeds automatically, expert users should always have the possibility to supply a curated set of seeds.

3.4. Crawling Modes

The crawling system is designed to be flexible and adaptable to various information needs. Specifically, it offers two primary modes of operation: Named Entity Crawling and Entity Type Crawling. These modes allow users to refine their search parameters, targeting either a specific entity or a broader category of entities. The choice of mode significantly impacts the crawling strategy and the algorithm for calculating relevance scores. By offering these two modes, the system provides the user with the flexibility to tailor the crawling operation according to specific needs. The choice of the crawling mode should determine the relevance metrics that are applied to crawled pages.

3.4.1. Named Entity Crawling

In this mode, the crawler is programmed to target a specific entity, as determined by the user. The user must provide two essential pieces of information: the name of the entity and its type. For instance, if a user is interested in the "Red Hot Chili Peppers" under the type "MusicGroup," the system will focus its crawling efforts on this specific entity. To accomplish this, a dedicated relevance metric should be calculated for the provided entity name. This metric considers factors such as keyword frequency, contextual relevance, and link analysis within the domain of the chosen entity type. This ensures that the crawled pages are not just related to the general type but specifically to the named entity.

3.4.2. Entity Type Crawling

In the entity type crawling mode, the system's focus is not limited to a specific entity but extends to all entities that share a particular schema.org type, such as "MusicGroup." In this mode, the crawler would collect data on multiple entities, like the "Red Hot Chili Peppers", the "London Symphony Orchestra", and "AC/DC". In this mode, the relevance metric for entity names cannot be applied, but besides of this difference, the relevance calculation should be identical to the other crawling mode.

3.5. Relevance Metrics

The core challenge of this project is to design a relevance metric for web pages that reflects whether a visited web page could be relevant for the information need that is coded in the user's query. The metrics should differentiate between the actual HTML content on a page (visible for end users on a web browser) and the structured data that is present on the page (in the form of Microdata or JSON-LD), because it allows for a more nuanced understanding of a page's content and the processing of those distinct types of data presentation differs in the FactCheck Framework. While structured data is easy to extract from web pages because of its standardized format, extracting information from plain text or HTML encoded content imposes several challenges. Information on the web is mostly unstructured or semi-structured. According to Eberendu et al. [25], "Semi-structured data is irregular data that may be incomplete and have a structure that changes rapidly or unpredictably but does not conform to a fixed or explicit schema"; some examples of semi-

structured data on the web would be HTML tables and lists. Unstructured data is not organized in any pre-defined manner, mainly free text.

To differentiate between structured and un- or semi-structured data, the decision was made to introduce two relevance metrics, a Structured Data Score, and a HTML Score.

3.5.1. Structured Data Score

The decision whether structured data on a web page contains information about the queried entity or entity type is straight forward. Structured data can be extracted in the form of semantic triples. A semantic triple, or RDF triple, is a set of three values that codes a statement – or fact – in the form of a subject-predicate-object expression. Some web sites use this format to describe their content in a machine-readable way, mostly to help search engines to understand and make use of this content. Based on the set of triples that can be extracted from a web page, the Structured Data Score is calculated; it is denoted by the number of triples that have the desired entity as subject OR object, divided by the number of all triples on the web page.

$$SD\ Score = \frac{N_{relevant\ triples}}{N_{all\ triples}}$$

The definition of "desired entity" differs depending on the mode the crawler is operating in. When the system is configured to crawl for a named entity, the desired entity must match the user supplied entity name as well as the user supplied entity type. When the System is crawling for an entity type only, the desired entity only has to match the supplied entity type, which means that one web page can contain multiple "desired entities". During the implementation and testing of this score calculation, it became evident that web pages tend to contain many triples (median of 100 triples per page in the first dataset collected for the experimental evaluation), addressing a multitude of entities and mostly about the web page itself. With the calculation method mentioned before, this leads to low scores, even if the page contains valuable structured data about the queried entity (type). This observation lead to the decision to apply a logarithmic scaling function to the base Structured Data Score. This function affects lower scores stronger than higher scores, while keeping the original range from zero to one.

$$SD\ Score_{final} = \frac{\log\left(1 + 100 * SD\ Score\right)}{\log\left(1 + 100\right)}$$

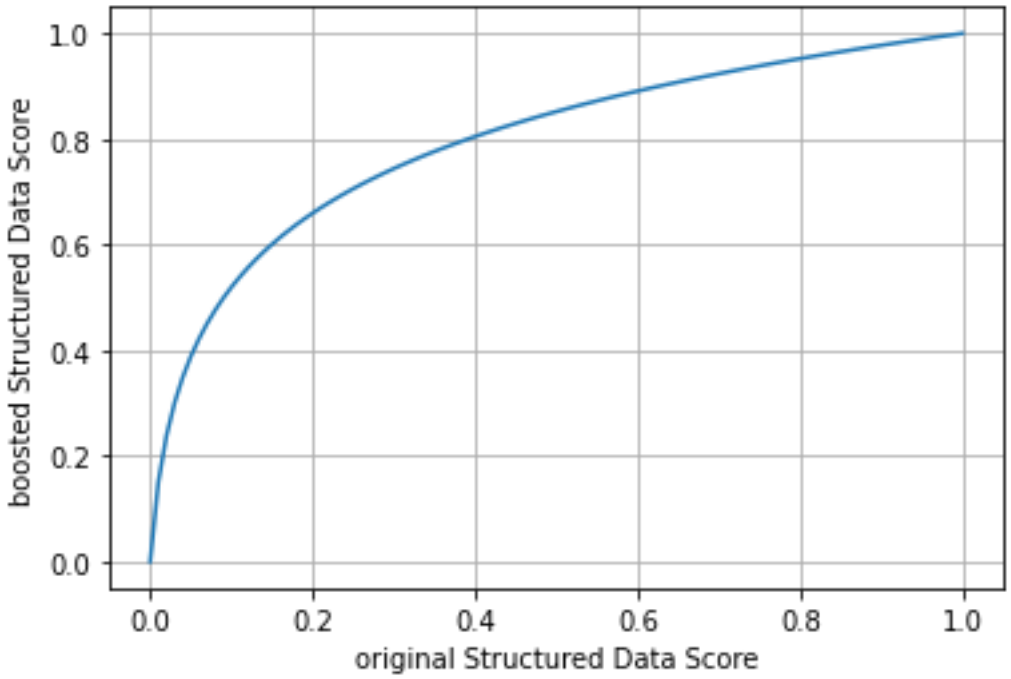


Figure 3.1.: Plot of the logarithmic scaling function applied to the Structured Data Score

3.5.2. HTML Score

The decision whether the unstructured or semi-structured data – in the form of free text, tables, or lists – on a web page contains information about the queried entity or entity type is more delicate, as the unstructured nature of the data imposes two main challenges on the processing of the data:

1. **Language –** Text on the web can come in virtually any language. In most cases, the language of a web page is indicated by a dedicated HTML tag, so language detection is not a problem. The problem lies within the understanding of different languages. Because implementing dedicated solutions for each and every language is not feasible in the scope of this project, the proof-of-concept implementation of the crawling system should focus on web pages in English language, as it is the most used language on the indexed web with an estimated 53.9% share [26]. The implementation should provide the possibility to extend the functionality for other languages in future development iterations.
2. **Ambiguity –** Entities can have alter egos, nicknames, or different spellings of their name. Also, various entities can share the same name, e.g., there is a person named "Steven Paul Jobs" who also goes by "Steve Jobs". Then there is a movie named after the person, also called "Steve Jobs". The same applies for properties of entities: There are multiple ways to express that an entity has a specific property, e.g., "Steve Jobs was born in 1955", "Jobs' birthday is February the 24.", or in a semi-structured manner as part of a table: "Date of Birth: February 24, 1955". Furthermore, web pages can contain nested information on multiple entities. For

example, a page that contains the biography of Steve Jobs could include a sentence like “In 1991, Jobs got married to Laurene Powell, a 28-year-old businesswoman.”.

To deal with these challenges, multiple metrics were designed, that cover various features of HTML documents that can help identifying entities. These metrics consider the structure of HTML pages, as well as their content. To achieve a final score in the range of zero to one, all of the following sub scores are normalized.

1. **Entity Name Score** – This score should reflect, how well a web page’s content matches the queried entity’s name. To achieve this, we check the frequency of HTML paragraphs (`<p>` tags) and headings (`<h1>` - `<h6>` tags) that contain the entity’s name (sub score S_1). Two other important properties of web pages are considered in this metric; the HTML `<title>` tag (reflected by sub score S_2), and the URL of the page (reflected by sub score S_3). Those properties are commonly used to evaluate web page relevance in search engines [27], [28], [29]. In all three cases, we look for case insensitive matches. If the entity name consists of multiple words, separated by whitespace, we check if each of the words occurs in the element’s text. This is done because in URLs, whitespace characters are often replaced by other characters which could lead to false negatives. The Entity Name Score is defined as the arithmetic mean of the three sub scores:

$$Entity\ Name\ Score = \frac{S_1 + S_2 + S_3}{3}$$

Where

$$S_1 = \frac{N_{<h>\ \&<p>\ containing\ name}}{N_{<h>\ \&<p>}}$$

$$S_2 = \begin{cases} 1 & if\ title\ contains\ query \\ 0 & otherwise \end{cases}$$

$$S_3 = \begin{cases} 1 & if\ URL\ contains\ query \\ 0 & otherwise \end{cases}$$

Entity Name Score also fulfills a gatekeeper function. If this score evaluates to zero, it is to be assumed that the web page does not address the queried entity. This also helps to reduce cost in terms of computation resources, as other scores do not have to be calculated if Entity Name Score is zero. Of course, this score should only be calculated in crawling sessions that are configured to focus on a named entity, and not only on an entity type.

2. **Properties Score** – This score should reflect how many properties of the queried entity type are present on a web page. To achieve this, we count the number of schema.org properties of the given type that are present on the page and divide that number by the number of all properties that are assigned to the type. A property is considered as present on the page, if the property itself or any of its synonyms are present on the page, multiple occurrences of the same properties are neglected:

$$Properties\ Score = \frac{N_{present\ properties}}{N_{all\ properties}}$$

3. **Entity Type Score** – This score should reflect how well the content on a web page matches the queried entity type. To achieve this, we employ a similar strategy as for the Entity Name Score; we calculate the ratio of paragraphs and headings that contain the name of the entity type ($N_{<h>\ \&<p>\ containing\ type}$) to all paragraphs and headings on a page ($N_{<h>\ \&<p>}$). A paragraph or heading is considered a match, if the entity name itself or one of its synonyms is present:

$$Entity\ Type\ Score = \frac{N_{<h>\ \&<p>\ containing\ type}}{N_{<h>\ \&<p>}}$$

4. **ChatGPT Score** – Currently, Large Language Models (LLM) are heavily used in various fields to investigate their potential for different applications. Especially the rise of ChatGPT[1], developed by OpenAI[2] has brought attention to LLMs. In the field of NLP, ChatGPT has shown to be well-suited for text classification tasks due to its large-scale pre-training [30]. To investigate the usefulness of LLMs for this particular use case, different systems, models, and prompts were evaluated.

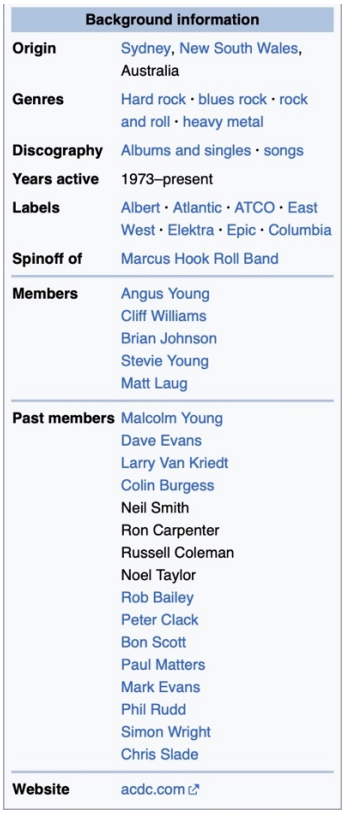
   The ChatGPT REST-API is a paid service, cost depends on the amount of text that is processed. Also, the API is rate-limited at 90.000 tokens (one token consists approx. of four characters) per minute, and the used gpt-3.5-turbo-0613 model has a maximum context size of 4096 tokens. For these reasons, the data that is passed to ChatGPT has to be carefully selected. Simply letting ChatGPT process all text from every crawled web page would not be feasible, as the API's rate limit would have a massive impact on the crawl-rate (processed pages per minute), and the cost for a crawling session could be significantly increased. As the previous scores primarily focus on the free text contained in web pages, and neglect semi-structured information in the form of HTML tables and lists, the ChatGPT score is focused on those data. The semi-structured data on a web page is of particular interest for the FactCheck++ framework, as it is easier to convert into key-value pairs – or facts.

   For each web page, all HTML table and list tags (`<table>`, `<ul>`, `<ol>`, `<dl>`) are extracted from the pre-cleaned DOM. If their text is longer than the maximum context of 4096 tokens, it is truncated to match that number. Then, a request is sent to the ChatGPT API, including the following system context:

   > *You are a useful tool for analyzing text. Your purpose is to evaluate whether a text snippet contains information about a **ENTITY_TYPE**. You return a score between 0 and 1 and NOTHING else. that score reflects your confidence that the supplied text contains information about a **ENTITY_TYPE**. If no information is present, you return 0.*

[1] https://openai.com/chatgpt
[2] https://openai.com

Here, ***ENTITY_TYPE*** is replaced by the actual entity type the system is crawling for. This system context was chosen based on initial experimental tests that were carried out on the OpenAI playground[1], which is a useful tool for testing the API's models and functions in a user-friendly way. Multiple versions of the system context were tested, always providing the same data to be analyzed by the chatbot. The data used was copied from info boxes of Wikipedia articles about different entities, namely the rock band AC/DC [31], the basketball team Los Angeles Lakers [32], and the movie A Clockwork Orange [33].



Background information	
Origin	Sydney, New South Wales, Australia
Genres	Hard rock · blues rock · rock and roll · heavy metal
Discography	Albums and singles · songs
Years active	1973–present
Labels	Albert · Atlantic · ATCO · East West · Elektra · Epic · Columbia
Spinoff of	Marcus Hook Roll Band
Members	Angus Young Cliff Williams Brian Johnson Stevie Young Matt Laug
Past members	Malcolm Young Dave Evans Larry Van Kriedt Colin Burgess Neil Smith Ron Carpenter Russell Coleman Noel Taylor Rob Bailey Peter Clack Bon Scott Paul Matters Mark Evans Phil Rudd Simon Wright Chris Slade
Website	acdc.com

Figure 3.2.: Wikipedia info box of the rock band AC/DC

Conference	Western
Division	Pacific
Founded	1947
History	**Minneapolis Lakers** 1947–1948 (NBL) 1948–1960 (NBA) **Los Angeles Lakers** 1960–present[1][2][3]
Arena	Crypto.com Arena
Location	Los Angeles, California
Team colors	Purple, gold, black[4][5][6]
Main sponsor	CJ CheilJedang[7]
President	Jeanie Buss
General manager	Rob Pelinka
Head coach	Darvin Ham
Ownership	Buss Family Trusts (majority)[8] Jeanie Buss (controlling owner) Philip Anschutz, Edward P. Roski, and Patrick Soon-Shiong (minority)
Affiliation(s)	South Bay Lakers
Championships	**17** (1949, 1950, 1952, 1953, 1954, 1972, 1980, 1982, 1985, 1987, 1988, 2000, 2001, 2002, 2009, 2010, 2020)
Conference titles	**19** (1972, 1973, 1980, 1982, 1983, 1984, 1985, 1987, 1988, 1989, 1991, 2000, 2001, 2002, 2004, 2008, 2009, 2010, 2020)
Division titles	**33** (1950, 1951, 1953, 1954, 1962, 1963, 1965, 1966, 1969, 1971, 1972, 1973, 1974, 1977, 1980, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 2000, 2001, 2004, 2008, 2009, 2010, 2011, 2012, 2020)
Retired numbers	**13** (8, 13, 16, 22, 24, 25, 32, 33, 34, 42, 44, 52, 99)
Website	www.nba.com/lakers

Figure 3.3.: Wikipedia info box of the Los Angeles Lakers

Directed by	Stanley Kubrick
Screenplay by	Stanley Kubrick
Based on	*A Clockwork Orange* by Anthony Burgess
Produced by	Stanley Kubrick
Starring	Malcolm McDowell Patrick Magee Adrienne Corri Miriam Karlin
Cinematography	John Alcott
Edited by	Bill Butler
Music by	Wendy Carlos[a]
Production companies	Polaris Productions Hawk Films
Distributed by	Warner Bros. (US) Columbia-Warner Distributors (UK)
Release dates	19 December 1971 (New York City) 13 January 1972 (United Kingdom[1]) 2 February 1972 (United States)
Running time	136 minutes[2]
Countries	United Kingdom[3] United States[3]
Language	English
Budget	$1.3 million[4]
Box office	$114 million[4]

Figure 3.4.: Wikipedia info box for the movie A Clockwork Orange

For example, an earlier version of the system context looked like this:

> *You are a useful tool for extracting properties of the schema.org type **ENTITY_TYPE** from text. For each message you receive, you return a score between 0 and 1 that reflects the relevance of the message's text for the schema.org type **ENTITY_TYPE**. The more properties can be extracted from the text, the higher is the returned score. Your answer ALWAYS begins with the score, and you provide ONLY the score. If you cannot extract properties of the schema.org type **ENTITY_TYPE** from the text, you return 0.*

[1] https://platform.openai.com/playground

This system context is noticeably longer, taking up around 110 tokens (depending on the length of the provided entity type), in contrast to the 69 tokens of the final system context. During the first experiments, it was observed that longer versions of the context would result in longer answers, in which the chatbot would try to justify the assigned score. To deal with this, the system context was shortened in an iterative process, trying to keep as much information in the text as possible. To check whether the chatbot would respond with false positives, ***ENTITY_TYPE*** was changed without changing the provided info box data, which the chatbot handled well by returning low scores in those scenarios.

5. **Combined Score** – Originally, the final HTML Score was defined as the mean of the four sub scores, including the gatekeeper function of Entity Name Score:

$$HTML\ Score_{basic} = \begin{cases} \dfrac{S_{Name} + S_{Properties} + S_{Type} + S_{ChatGPT}}{4} & if\ S_{Name} > 0 \\ 0 & otherwise \end{cases}$$

The experimental evaluation of this approach showed that while the HTML Score allows for distinguishing relevant from irrelevant pages, the scores tend to be relatively low. For that reason, a sigmoid function was designed and applied to the final HTML Score, such that it covers the entire range from zero to one. An in-depth explanation of the evaluation and design of this function and its purpose is provided in chapter 5. With the sigmoid function applied, the final HTML score is calculated as shown below:

$$HTML\ Score_{final} = \frac{1}{1 + e^{-20\ (HTML\ Score_{basic} - 0.3)}}$$

4. Design and Prototypical Implementation

This chapter covers the design of the software prototype, as well as considerations regarding its implementation, including information on the system's architecture, its workflow and APIs used by the system.

4.1. System Architecture and Technology Stack

The implementation of the crawling system consists of two main components: a frontend and a backend application, both written in Java and using the Spring framework[1]. Spring is a comprehensive and modular framework for building enterprise-level applications in Java. The framework focuses on providing the tools and infrastructure to facilitate the efficient development of secure, robust, and scalable applications.

The frontend application follows the classic MVC (Model-View-Controller) design pattern, while the crawler – or backend – application follows the Master/Slave design pattern, which is commonly used in multithreaded applications. The two applications share a library component that holds common code used by both applications. This includes the database abstraction, various Spring configurations and utility classes. Both applications share a central PostgreSQL[2] database, providing a single, consistent view of data for both frontend and backend applications. A single database also ensures data consistency, integrity, and reduced data redundancy.

For database abstraction, JPA (Jakarta Persistence API) and Hibernate ORM was used. Hibernate is an implementation of JPA that provides mechanisms to perform CRUD (Create, Read, Update, Delete) operations without the need to write boilerplate code.

The frontend and backend communicate via RabbitMQ[3]; an open-source message broker that uses AMQP (Advanced Message Queuing Protocol) for messaging. This promotes loose coupling between both components and provides effortless horizontal scaling possibilities for future use cases, as outlined in section 7.2.

Structured data extraction was implemented using the Apache Any23[4] (Anything to Triples) library. It can extract data from HTML tags, Microformats, RDFa, JSON-LD, or Microdata and merge those data in a common format like Turtle (Terse RDF Triple Language), which makes it easier to process in subsequent steps.

The application can be built and deployed using Docker[5] technology. Docker Compose[6] is used to package all components of the system in a single container, which allows deployment on a single machine with a single command. The resulting deployment includes four Docker containers (PostgreSQL database, RabbitMQ message broker, crawler application and the server application for the REST-API and the web interface) that are linked via a Docker network.

[1] https://spring.io
[2] https://www.postgresql.org
[3] https://www.rabbitmq.com
[4] https://any23.apache.org
[5] https://www.docker.com
[6] https://docs.docker.com/compose/

For tracking changes in the source code and version control, Git[1] was used. The project repository was created on the GitLab[2] Server of the University of Vienna.

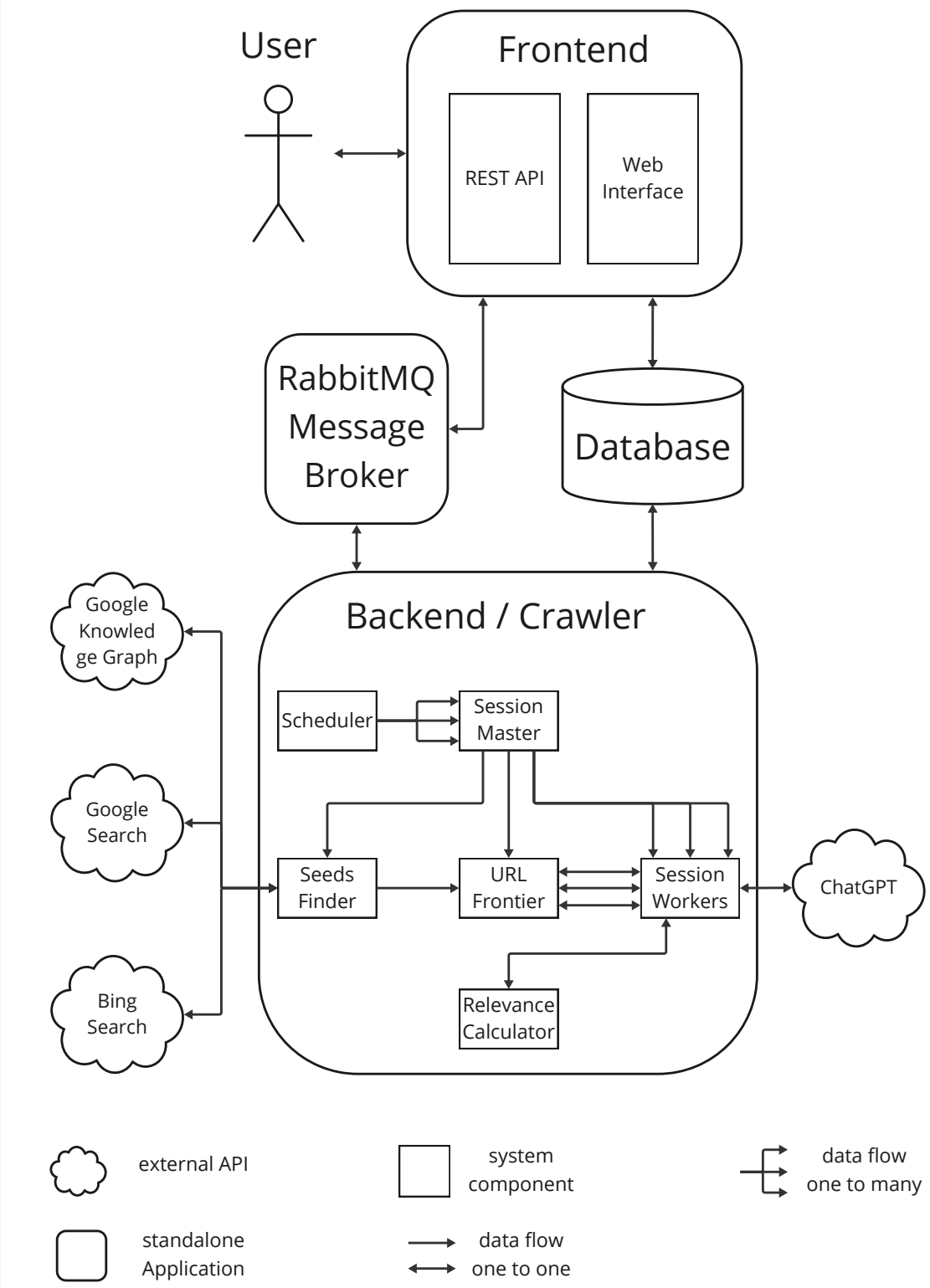


Figure 4.1.: Simplified system overview

[1] https://git-scm.com
[2] https://about.gitlab.com

4.2. Data Model

All data that is used and collected by the system is persisted in a central PostgreSQL database. In total, there are eight tables that represent the data model of the system:

- **Sessions Table –** This is the central table of the system. It holds all input parameters of crawling sessions, as well as some metadata for each session (e.g., number of pages that have been visited, date and time of the session, session status, number of failed requests)
- **Domains Table –** Each web domains base URL that has been encountered by the crawler is stored here, along with some metadata (URLs that are not allowed by the domains robots.txt file, number of visited pages on the domain, and whether the domain is blacklisted)
- **Pages Table –** Each web page that has been visited by the crawler has an entry in this table. Along with its URL, the date and time of the last visit is stored.
- **Pages_R_Sessions Table –** This is a relation table that represents a One-to-Many relationship between sessions and pages (one page can be visited in many sessions). It stores the relevance metrics for a page in the context of a session (the degree of relevance of a page depends on the query).
- **Session_Status Table –** This table stores the possible states a session can be in: “Pending”, “Running”, “Cancelling”, “Finished”, “Terminated”.
- **Schema_Types Table –** Here, every Schema.org entity type is stored, along with its “humanized” name (Schema.org uses camel case in its nomenclature) and english synonyms.
- **Schema_Properties Table –** This table holds all Schema.org properties, “humanized” property names and synonyms, corresponding to the Schema_Types table.
- **Types_R_Properties Table –** This is a relation table that represents a Many-to-Many relationship between Schema.org types and properties (a property can belong to multiple types and a type has multiple properties).

These tables create a normalized and efficient data model that allows for effective data storage and retrieval, ensuring data integrity and consistency. The use of foreign keys to establish relationships between tables enables complex queries which are helpful for finding seeds for the crawler based on historic crawling sessions, as well as for data analysis. The relations between the tables are illustrated in Entity Relationship Diagram (Figure 4.2).

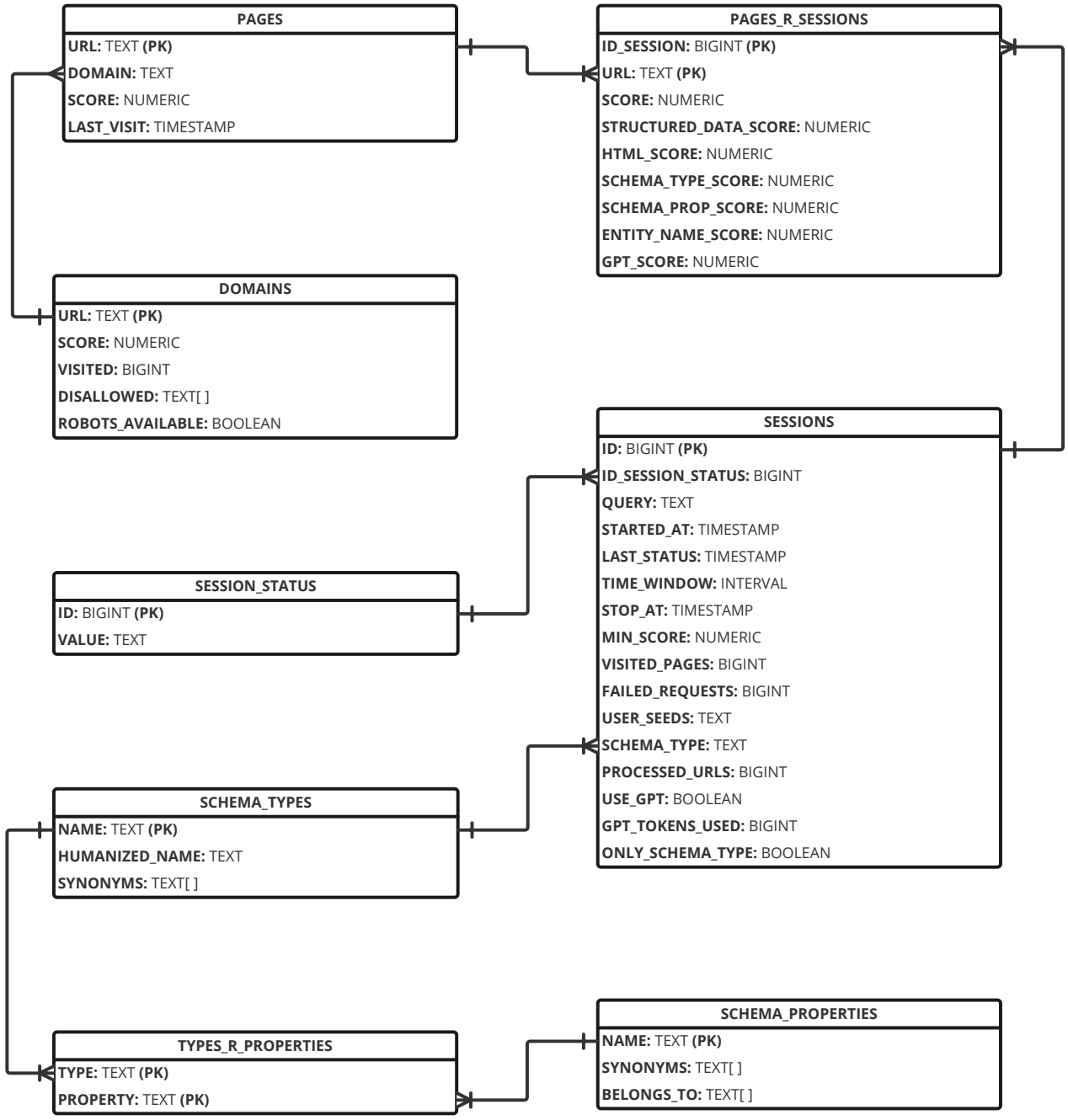


Figure 4.2.: Entity Relationship Diagram of the systems data model

Hibernate ORM[1] is used to bridge the gap between the object-oriented Java code and the relational Postgres database. Hibernate allows to map Java classes to database tables and Java data types to SQL data types, abstracting the underlying database operations. This mapping is done for all the above tables, simplifying the interactions with the database.

[1] https://hibernate.org/orm/

4.3. Integrated APIs

To enable the system to acquire high quality seeds and to support the relevance estimation of web pages, several third-party APIs are integrated in the crawling system. These APIs are used by the Crawler Application to acquire seeds (SeedsFinder component) and to help estimating the relevance of visited pages (RelevanceCalculator component).

4.3.1. Google Knowledge Graph Search API

A knowledge graph represents a collection of interlinked descriptions of entities in a structured form. It aims to store factual information and capture the relationships between these entities. Knowledge graphs are typically visualized as nodes (representing entities) and edges (representing relationships). By structuring data in this way, it is possible to make semantic queries and understand the context in which these entities exist.

The Google Knowledge Graph Search API allows developers to retrieve data – in this case, we are mostly interested in URLs – about entities from the Google Knowledge Graph by providing a query and entity type. As the Google Knowledge Graph uses standard schema.org entity types to organize its entities, it matches well with the proposed system.

In the design phase of the project, it was planned to use the Google Knowledge Graph API for acquiring seeds in both crawl modes of the system – crawling for a single named entity, or an entity type. During the development, it turned out that the Google Knowledge Graph API is only suitable for the latter. This is because the API tends to return information on related entities when a specific entity is requested. For example, a request with the query "Angela Merkel" for the entity type "Person" returns information on Angela Merkel, but also on other german politicians like Annalena Baerbock, Friedrich Merz, or Markus Söder. This is not very helpful for the purpose of acquiring high-quality seeds for the specific entity Angela Merkel, as the URLs attached to those other entities can cause the crawler to follow paths that do not lead to relevant pages about Angela Merkel.

In the case of the other crawl mode, this can be helpful, as the API responses contain information on many different entities of the same type, diversifying the initial seeds set. Because of this, the decision was made to only employ the Google Knowledge Graph API for crawling sessions that only crawl for an entity type. The API requires the "query" parameter to be present in the request, which is why the parameter is set to the wildcard "*" on every request.

The API returns a list of results in JSON format, along with some additional data on the initial query and the results. The API is free to use and requires a Google API key. For a request for the entity type "Person", a typical result element looks like this:

```
{
    "@type": [Thing", Person"],
    "name": "Phil Hartman",
    "detailedDescription": {
        "articleBody": "Philip Edward Hartman was a … graphic
designer.",
        "url": "https://en.wikipedia.org/wiki/Phil_Hartman"
    },
    "description": "",
    "@id": "kg:/m/03d_zl4",
 }
```

Listing 4.1.: Example result element of the Google Search API in JSON format

4.3.2. Google Custom Search API

The Google Custom Search API serves as a crucial component within the SeedsFinder subsystem of the focused web crawling system. Its primary function is to programmatically query Google's search engine and fetch initial seed URLs, which the crawler uses as starting points for both Named Entity Crawling and Entity Type Crawling modes.

The API returns results in JSON format, making it convenient to parse and integrate into the system. Unlike the Google Knowledge Graph API, which is preferred for the entity type crawling mode, Google Search yields better search results when the query contains the name of a specific entity name. The results returned when only searching for an entity type have nearly no value for the use case of the crawler. For example, the query “person” would almost exclusively return links to dictionaries explaining the word “person”, which would lead the crawler to visit irrelevant pages. This why the Google Search API was chosen to be used only for the named entity crawling mode. A request with the query “Angela Merkel” returns a list with elements that look like this:


```
{
    "kind": "customsearch#result",
    "title": "Angela Merkel - Wikipedia",
    "htmlTitle": "<b>Angela Merkel</b> - Wikipedia",
    "link": "https://en.wikipedia.org/wiki/Angela_Merkel",
    "displayLink": "en.wikipedia.org",
    "snippet": "Merkel was [...] the most powerful woman in the world.",
    "formattedUrl": "https://en.wikipedia.org/wiki/Angela_Merkel",
    "pagemap": {
        "hcard": [
            {
                "url_text": "Official website",
                "bday": "1954-07-17",
                "fn": "Angela Merkel",
                "nickname": "Angela Dorothea Kasner",
                "label": "Am Kupfergraben, Berlin",
                "url": "Official website"
            }
        ],
        "metatags": [
            {
                "referrer": "origin",
                "og:image":
"https://wikimedia.org/[...]/Merkel_2019.jpg",
                "og:image:width": "1200",
                "og:type": "website",
                "og:title": "Angela Merkel - Wikipedia",
                "og:image:height": "1678",
                "format-detection": "telephone=no"
            }
        ]
    }
}
```


Listing 4.2.: Example result element of the Google Custom Search API in JSON format

4.3.3. Bing Search API

Similar to the Google API, Bing provides a REST-API that allows access to the search engine in a programmatic way. Bing is also used in the SeedsFinder component to complement the search results of Google, and to increase diversity in search results. It is

exclusively used in the named entity crawling mode for the same reasons mentioned in the previous section. The API is free to use and requires a Bing API key. A typical search result element for the query “Angela Merkel” looks like this:


```
{
    "id": "https://api.bing.microsoft.com/api/v7/#WebPages.6",
    "name": " Neuste Nachrichten zur Bundeskanzlerin - Handelsblatt",
    "url": "https://www.handelsblatt.com/themen/angela-merkel",
    "isFamilyFriendly": true,
    "displayUrl": "https://www.handelsblatt.com/themen/angela-merkel",
    "snippet": "Merkel ist die erste Bundeskanzlerin Deutschlands.",
    "dateLastCrawled": "2023-09-04T21:29:00.0000000Z",
    "language": "de",
    "isNavigational": false
}
```


Listing 4.3.: Example result element of the Bing Search API in JSON format

4.3.4. ChatGPT API

OpenAI provides a multipurpose REST-API that allows developers to use and fine-tune different language models. The API also allows to chat with OpenAI’s famous chatbot, ChatGPT. It is important to note that the use of OpenAI’s REST-API carries financial implications, operating on a pay-per-use model. In the context of this project, the API is used to pass website data to the chatbot and let it estimate the relevance of the provided text regarding a given entity type. Details on how the API is used by the crawling system are provided in 4.4.10. Data Processing & Relevance Computation

4.4. Crawler

The Crawler application is the core component of the proposed system. It holds the Scheduler, SeedsFinder, SessionMaster, SessionWorker, RelevanceCalculator, and URL Frontier subcomponents which are responsible for the crawling process. The following chapters provide an in-depth explanation of each of these subcomponents, their functionalities, and how they interplay to ensure a seamless and efficient crawling process. Figure 4.3.: Hierarchic structure of the crawlershows a simplified tree-view of the crawler’s subcomponents. The application runs a single Scheduler instance, that acts as a gateway for incoming crawling requests. For each incoming request, a dedicated SessionMaster thread is started, that is responsible for managing a single crawling session. A SessionMaster manages a single URL Frontier, a single SeedsFinder, and up to 50 SessionWorker threads. A SessionWorker manages a JsoupUtil instance, which acts as an abstraction layer for fetching and processing web pages. The RelevanceCalculator component is responsible for calculating relevance metrics in a multithreaded manner, splitting the relevance calculation task into two subtasks: one for calculating the HTML Score of a web page, and one for calculating the Structured Data Score.

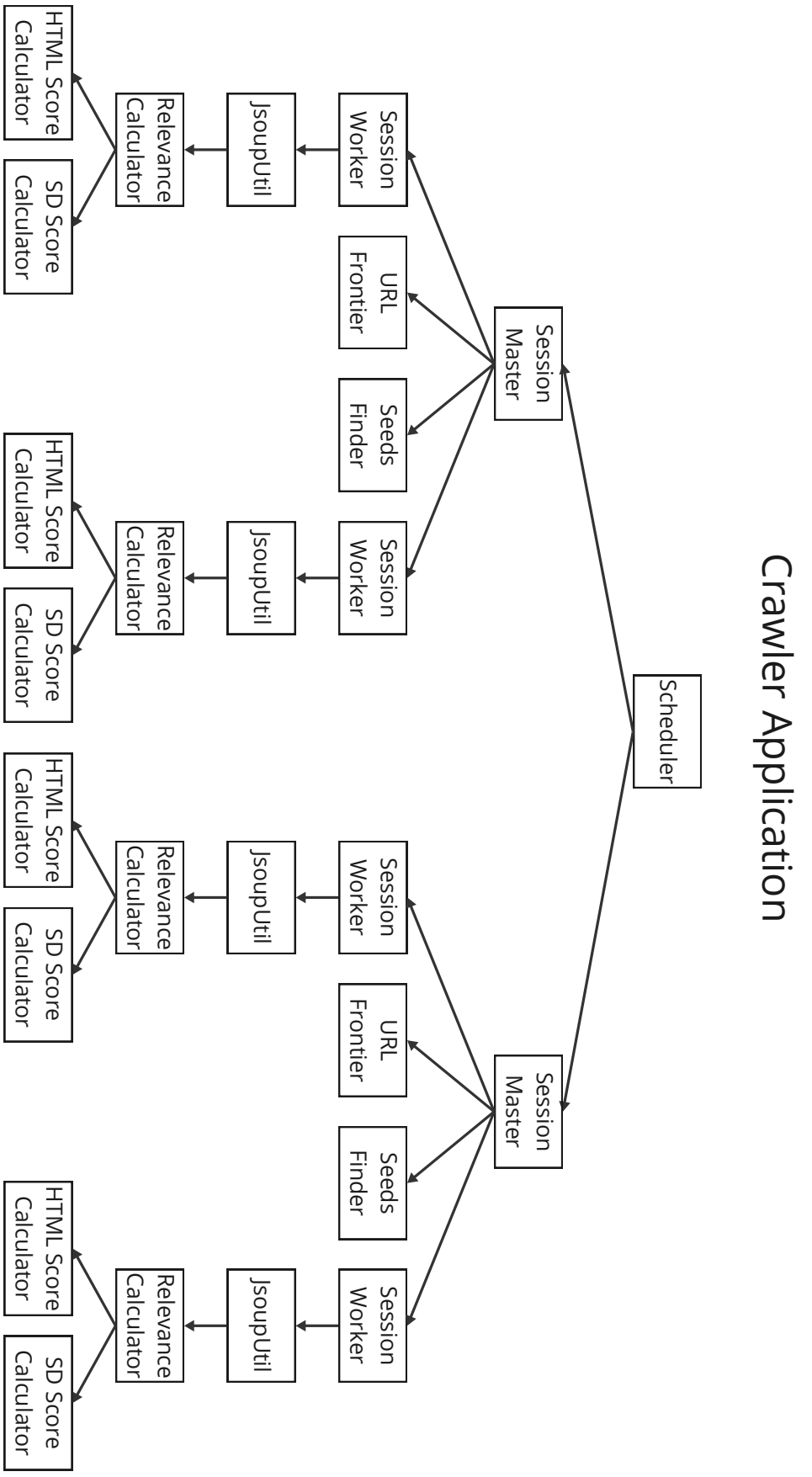


Figure 4.3.: Hierarchic structure of the crawlers components

4.4.1. System Workflow

The workflow of the crawling system is depicted in the flowchart below. This section provides a more in-depth explanation of the steps that are carried out in the crawling process.

As soon as a new crawling session is triggered via the frontend application (either via the REST-API or via the web interface), the Scheduler component of the crawler receives a message via a dedicated RabbitMQ queue, that holds the database ID of the new session.

The Scheduler then initializes a new SessionMaster instance, passing the ID of the session as a parameter. This instance is cached by the Scheduler in a HashMap, to be able to communicate with the SessionMaster during execution. The SessionMaster component then prepares the crawling process. It sets the session status to “running”, which also means that the user-supplied time frame for the crawling session starts. The SessionMaster also tokenizes the queried entity name (the name is split into a list of strings, using space characters separators), and initializes a SeedsFinder instance.

The Seedsfinder component fetches seeds from multiple sources, including Google Knowledge Graph, Google Search, Bing Search, and the system’s database. Together with the seeds supplied by the user, these seeds are then used to initialize the URL Frontier, all with the highest priority.

The SessionMaster then starts multiple SessionWorker threads which are responsible for polling the URL Frontier, downloading web pages, and calculating their relevance metrics using the JSoupUtil component. While the SessionWorkers are downloading and processing web pages, the SessionMaster is in a kind of idle status, waiting for the SessionWorkers to complete and sending update pings to the frontend via a dedicated RabbitMQ queue. When the URL Frontier does not contain any URLs to process, or if the time frame configured by the user is over, the SessionWorkers terminate, indicating that the crawling session is over. The SessionMaster then takes over, updates the database representation of the crawling session a last time and terminates as well.

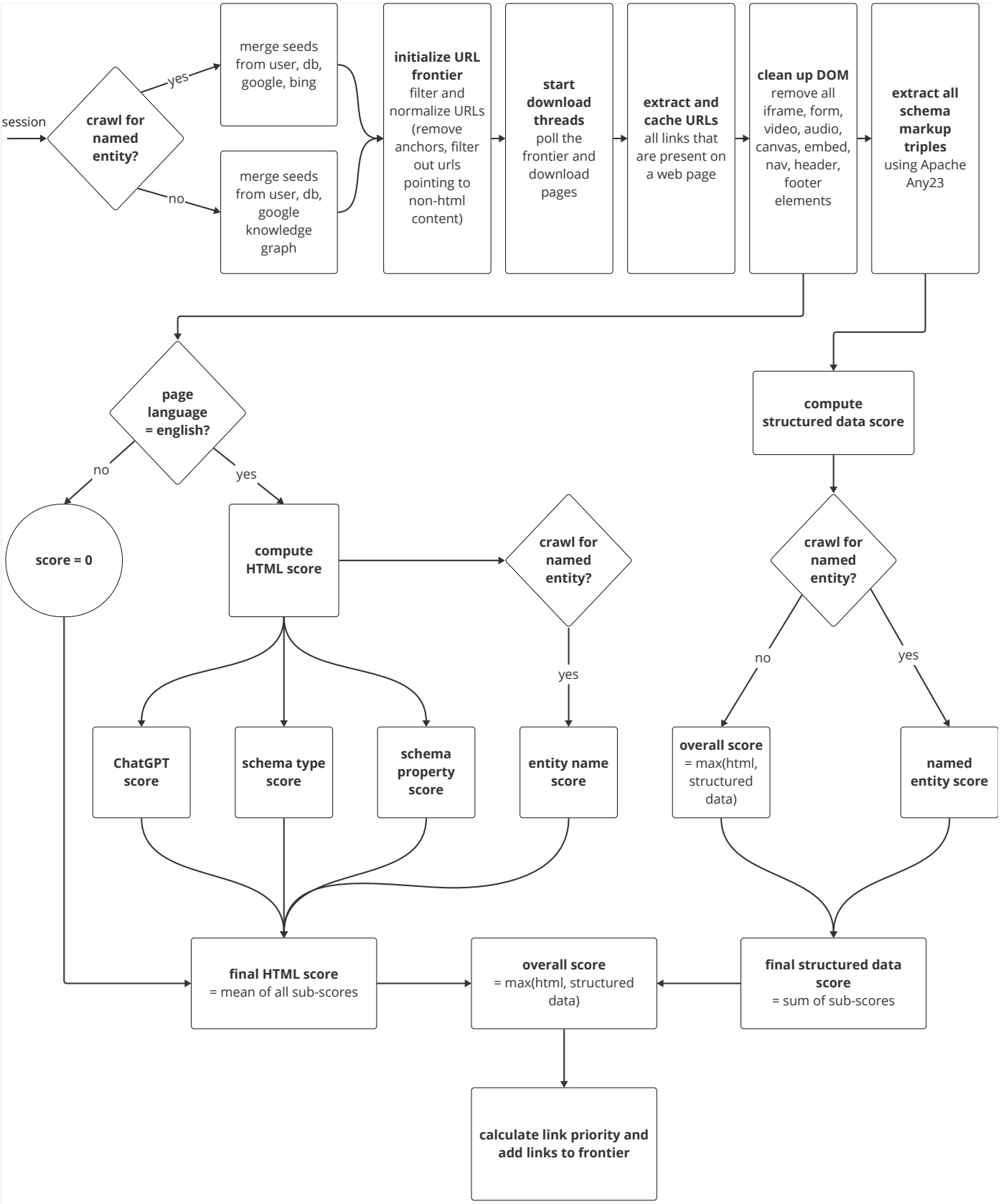


Figure 4.4.: Workflow of the crawler application

4.4.2. Scheduler Component

The Scheduler component is responsible for managing crawling sessions on the highest level. It is the only entry point for starting and terminating crawling sessions. The Scheduler caches all running crawling sessions, such that it can communicate with their main threads. The Scheduler also holds all instances to the application's JPA repositories, which are passed to a new SessionMaster instance, when a crawling session is triggered. The Scheduler class is annotated with Spring's @Component annotation[1], such that it can be automatically registered as a bean in the Spring application context. This allows for dependency injection and autowiring[2] of components.

4.4.3. SessionMaster Component

The SessionMaster is responsible for managing a single crawling session. It runs in a dedicated thread (it implements Java's Runnable interface) and runs all code that is necessary to start and end a crawling session. It updates the session status in the database, notifies the frontend of changes during a session via RabbitMQ, initializes the URL Frontier component and SessionWorker components, and triggers the SeedsFinder component. While a crawling session is running, the SessionMaster notifies the frontend application in five second intervals via a dedicated RabbitMQ queue, such that the frontend application itself can fetch data that was inserted into the database by the crawler. This way, the web frontend can stay updated, while the crawler and frontend components remain loosely coupled only by their connection to the message broker.

The SessionMaster class has no injected beans, as all necessary beans are autowired in the Scheduler component and passed to every SessionMaster instance as references.

4.4.4. SeedsFinder Component

As the name suggests, the SeedsFinder component is responsible for acquiring seed URLs for a given query. Depending on the crawling mode, the sources used to find seeds differ. The SeedsFinder uses external APIs to fetch seeds, as well as URLs submitted by the user and data from preceding crawling sessions to assemble a diverse set of high-quality seed URLs. The following sources are used by the SeedsFinder component:

Google Knowledge Graph was initially planned to be used in both crawling modes, as its API allows queries about named entities and entity types. During development, it was observed that the API tends to return data on various entities when queried for a single entity, as explained in 4.3.1. Google Knowledge Graph Search API Because of this, the decision was made to use the Google Knowledge Graph only for queries regarding an entity type. For this use case, the API is very helpful, as it returns a broad range of different entities of a type. Three parameters are submitted in each request, in addition to the Google API key:

[1] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html

[2] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html

- **Query –** This is a required parameter for which the API would expect an entity name, e.g., “Angela Merkel”. As we are not interested in searching for a single entity, we provide a wildcard value of “*”. This approach is not mentioned in Googles API documentation [34] but works just as intended.
- **Limit –** This parameter limits the number of results returned by the API. Its value is set to 300, while the maximum allowed value is 500. As stated in the API’s documentation, higher values for this parameter increase the risk of producing a request timeout and not receiving any data [34]. We are interested in getting many results from this API, because of the high diversity of its results. However, using the maximum allowed value of 500 indeed produced request timeouts on a regular basis, which is why the value was dialed down iteratively until timeouts did not occur anymore.
- **Types –** This parameter expects the name of at least one schema.org entity type, e.g., “Person” or “MusicGroup”. As the user provided entity type is directly taken from the schema.org definition, the value can be directly used in the request.

To extract all relevant URLs that are present in the result set returned from the API, the response is parsed using Google’s JSON parser library GSON[1], and then traversed extracting all “url” elements that have a “detailedDescription” parent.

As explained in section 4.3.2., Google Search is exclusively used in the named entity crawling mode. The API only allows a maximum of ten search results per request (promoting pagination), which is why in total, five requests are sent to the API to yield 50 search results from Google. Four parameters are passed to the API, besides the Google API key:

- **Q(uery) –** This parameter represents the search query. The value is set to the entity name supplied by the user, with the humanized version of the chosen entity type appended. This is done to promote disambiguation. For example, if the user configures the system to crawl for “Wacken” – a famous hard rock festival in Germany, named after the town it takes place in – of type “Festival”, the query submitted to Google would be “Wacken Festival”. This helps to reduce the number of links to pages about the town “Wacken” and increase the number of links that lead to relevant pages.
- **LR –** This parameter allows to filter the returned results by language. As the crawling system is currently only focusing on pages in English language, this parameter is set to “lang_en”. For future use cases covering multiple languages, this could be extended.
- **Start –** This parameter indicates the position of the first result to be returned. As mentioned before, the SeedsFinder submits five consecutive requests to the API, increasing this parameter by ten every time.

Bing Search serves the same purpose as Google Search in this context. Bing does not restrict the amount of search results that are returned by the API, but instead tends to return large numbers of results. To achieve a balance between seeds coming from Bing and

[1] https://github.com/google/gson

Google, the number of results is limited in the request. Besides the Microsoft API key, the following parameters are passed to the API:

- **Q(uery)** – This parameter represents the search query. Its value is set to the exact same as for the query parameter of the Google Search API.
- **Count** – This parameter controls the number of results that are returned by the API. It is set to 50.
- **ResponseFilter** – Bing allows to search for different kinds of things, including images or videos. Since we are only interested in web pages, this value is set to "Webpages".

Users have the possibility to manually provide seeds when configuring a crawling session. These seeds are expected to be of high quality and are merged with the other seeds without further processing.

The SeedsFinder component also acquires seeds from preceding crawling sessions, through the system's database. The exact query that is used depends on the current crawling mode. When crawling for an entity type, the system selects the top 100 URLs (based on their HTML and Structured Data Score) that were crawled in sessions that had ChatGPT scores enabled and focused a named entity of the given schema.org type. This is done because in sessions that focus a named entity, the crawler can achieve higher precision in the score calculation as an additional relevance metric based on the entity name is calculated in those sessions. The SQL for this query is shown below:

```
select s.url from (
    select
        distinct(p.url),
        greatest(p.html_score, p.structured_data_score) as g_score
    from pages_r_sessions p
    join sessions s on
        s.id = p.id_session and
        s.schema_type = :schemaType and
        s.query is not null and
        s.use_gpt = true
    order by g_score desc limit 100) s;
```

Listing 4.4.: SQL used to retrieve seed URLs for entity type crawling mode

When crawling for a named entity, the crawler selects the top 50 URLs that were crawled in sessions that focused the same named entity. The SQL for this query looks like this:

```
select s.url from (
    select
        distinct(p.url),
        greatest(p.html_score, p.structured_data_score) as g_score
    from pages_r_sessions p
    join sessions s on
        s.id = p.id_session and
        s.schema_type = :schemaType and
        s.query LIKE :query and
        s.use_gpt = true
    order by g_score desc limit 50) s;
```

Listing 4.5.: SQL used to retrieve seed URLs for named entity crawling

When all sources delivered their seeds, the SeedsFinder merges them into a single set, removing duplicate URLs and returns this set to the SessionMaster component, which then initializes the URL frontier with this set.

4.4.5. URL Frontier Component

The URL frontier is a critical component of any web crawling system. It determines the order in which URLs are visited by the crawler and hence influences the efficiency and effectiveness of the crawl. The URL frontier is implemented as a priority queue using Java's PriorityBlockingQueue class, which was chosen for the following reasons:

1. **Ordered Processing –** Control over the order of the queue useful in a focused crawler, where some URLs may be considered more relevant or important than others.
2. **Thread-Safety –** the PriorityBlockingQueue class is thread-safe, meaning it can be safely used by multiple threads without the need for additional synchronization. This is an essential feature for this system, as it involves multiple worker threads fetching and processing web pages concurrently.
3. **Efficiency –** this queue implementation offers efficient performance for insert and remove operations (logarithmic complexity). This leads to improved overall performance of the web crawler.
4. **Blocking Behavior –** as hinted by its name, a PriorityBlockingQueue has blocking capabilities. That means, if the queue is empty and a thread tries to poll an element, the thread will block until an element is available. This is particularly useful in a web crawler where worker threads may have to wait for new URLs to be discovered and added to the queue.

The priority of URLs is calculated based on the relevance of the source web page, the URL string itself and its anchor text. According to Batsakis et al., the combination of those two metrics for priority calculation yields a higher harvest rate (percentage of relevant pages returned), compared to other common priority metrics used in classic focused crawlers [35].

This calculation method is based on the assumption that relevant web pages tend to link to other relevant pages, which is empirically confirmed by a variety of studies, including [20] and [36].

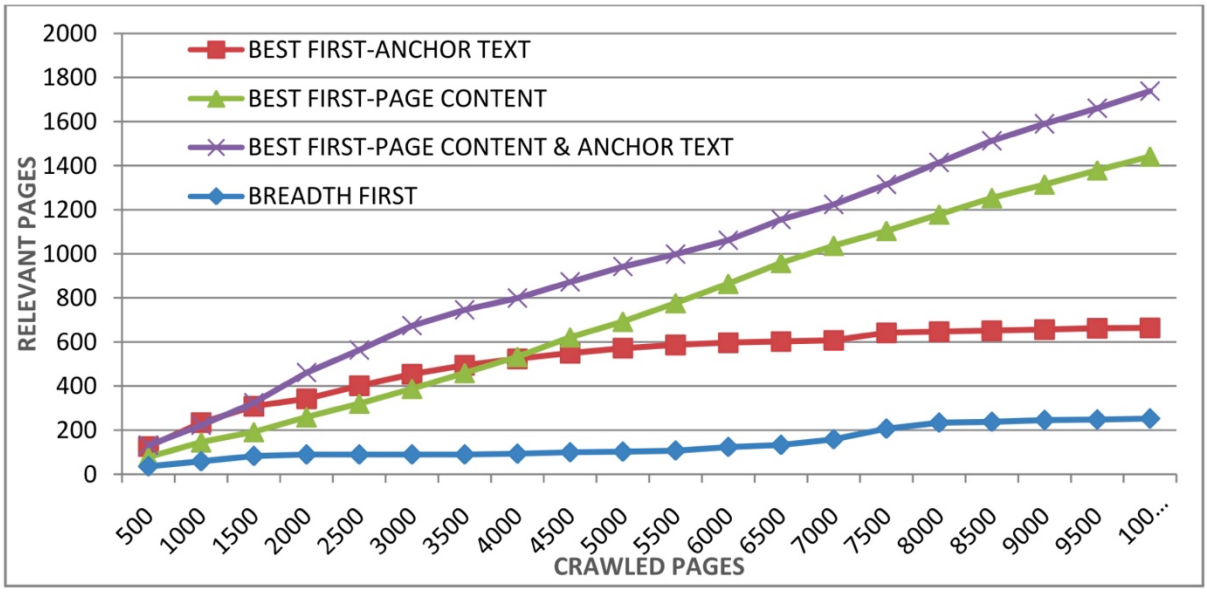


Figure 4.5.: Harvest rate comparison (Batsakis et al.)

Batsakis et al. also compared classic focused crawlers with learning crawlers that employ Hidden Markov Models (HMM) to learn the structure of relevant pages and paths to other relevant pages. In some cases, these crawlers can outperform classic focused crawlers in terms of harvest rate, in other cases, the classic focused crawler yields higher harvest rates. The authors assume that classic focused crawlers perform better on topics that can be unambiguously described by keywords [35]. The crawling system proposed in this work is mainly built for topics (or entities) that can be precisely identified by their name and type, so it is to assume that the use of an HMM is not beneficial for this use case. Furthermore, HMM crawlers come at the cost of lower download rates, as their relevance calculation method is more expensive in terms of computation resources. Of course, the size of this effect depends on the performance of the system running those operations, and could possibly be minimized by employing high-end computing resources, such that network load becomes the limiting factor [35].

During development, the URL frontier did not implement any sorting strategy, which means that the crawling system effectively followed a breadth-first strategy. In this stage, even when supplied with high quality seeds, the system was observed to quickly visit high numbers of irrelevant pages that contained even more links to irrelevant pages. This behavior contradicts one of the most important goals of this focused crawling system; to bypass irrelevant content and hence save on computing resources. After designing and implementing the priority-based approach, the benefit of evaluating link priority became obvious; the crawlers harvest rate significantly improved, as explained in 5.2. Method and Results

4.4.6. SessionWorker Component

The SessionWorker components are responsible for downloading web pages, parse their content and passing it to the RelevanceCalculater component. SessionWorkers run in

dedicated threads, to enable concurrent downloading and processing of web pages. A SessionWorker polls the URL frontier to get the next URL to process. The domain of the URL is then extracted to check whether the domain is already known by the system. For example, if the SessionWorker polls the URL https://en.wikipedia.org/wiki/Angela_Merkel from the frontier, the domain part of the URL would be https://en.wikipedia.org. If the domain is already present in the *domains* database table, the SessionWorker checks whether the URL contains a path that is disallowed by the domains robots.txt file. If the domain was never visited before, the SessionWorker tries to fetch and parse the domains robots.txt file using the RobotsUtil utility component. Then, the new domain is added to the database, together with the disallowed paths of the domain.

If the polled URL is not allowed to be crawled, the SessionWorker skips the current URL and fetches the next URL from the frontier. Otherwise, the SessionWorker initializes a new JsoupUtil instance passing the URL, which is then responsible for downloading the web page, extracting links from the page, and passing the pages content to a dedicated RelevanceCalculator instance.

4.4.7. RelevanceCalculator Component

The RelevanceCalculator's purpose is to return all relevance metrics (as defined in section 3.5) for a given web page. As the computation of these metrics is resource intensive, and external APIs are being employed introducing latency, the relevance calculation is multithreaded. By splitting this task into two subtasks (HTML Score calculation and Structured Data Score calculation), which are performed concurrently, the runtime of the relevance calculator is nearly cut in half. The RelevanceCalculator class defines methods to clean up a given HTML document and calculate relevance metrics based on the content and structure of those documents. The cleaning of documents is done with the help of the Jsoup library which allows to select HTML elements using CSS like selectors and remove them from the document in a single method call. The selector for cleaning the document is designed to match all HTML elements that are assumed not to be part of the main content of the given page, like headers, footers, and elements that purely serve the purpose of navigation. Details on the implementation of relevance metrics are illustrated in section 4.4.10.

4.4.8. JsoupUtil Component

For downloading and parsing web pages, Jsoup[1] is used. Jsoup is a Java library widely recognized for its powerful capabilities in working with HTML data. It implements the WHATWG HTML specification and allows for effective and efficient data extraction using DOM traversal. This means that it can navigate through the HTML tree structure and select certain elements based on their attributes, classes, or IDs, which is critical for a focused crawler that needs to locate and retrieve specific information from a web page. Programmatic DOM traversal is convenient with Jsoup as it supports CSS- or jQuery-like selectors.

[1] https://jsoup.org

Jsoup also implements a HTTP client that provides a high level of configurability and allows to parse a web page to its DOM representation by providing a URL in a single line of code.

The JsoupUtil component is implemented as a wrapper class for Jsoup that provides abstracted convenience methods that are used by the SessionWorker component. These methods include the caching of links on a page, relevance calculation for a page, and URL priority calculation for the cached links. This keeps the code in the SessionWorker component clean and readable, promoting code maintainability.

4.4.9. Politeness Policy

As stated in non-functional requirement 4, the crawler should adhere to web standards, such as the Robots Exclusion Protocol. The standard was originally proposed by Martijn Koster on the "www-talk mailing list" in 1994 [37], and quickly became the de facto standard to indicate areas on domains that are allowed or not allowed to be visited by web crawlers.

In September 2022, a proposed standard was published as RFC 9309 [38]. Following this protocol, site owners can place a text file – "robots.txt" – at the root of their site hierarchy (e.g., https://wikipedia.org/robots.txt), containing crawling rules in a specific format that can be understood by web crawlers. These rules rely on voluntary compliance. Wikipedia has an extensive list of known crawlers that do not comply with the standard in its robots.txt file [39].

During the development of the proposed crawling system, when its politeness policy was not yet implemented, it became evident that not obeying the rules defined in a robots.txt can lead to servers blocking requests from a crawler, which is usually preceded by HTTP responses with status code 429 – "Too Many Requests". To make the crawler compliant with the Robots Exclusion Protocol, a utility class was implemented that can parse a robots.txt file and returns all disallowed links on a domain. More precisely, the robots.txt tool seeks out the line "User-agent: *", which indicates, that the following lines are rules for every crawler or bot that is not specifically mentioned in the file. Then, the tool parses each line that starts with "Disallow:", e.g. "Disallow: /wiki/Category:Noindexed_pages". The tool then prepends the base URL of the domain to each entry and returns a list of all forbidden URLs on the domain. These URLs are then persisted in the database as an array in the domains table. If the tool finds a line "Disallow: *" or "Disallow: /", the entire domain is blacklisted and no page on this domain will be visited in current and future crawling sessions. If no robots.txt file is present at the root of the domains hierarchy, the system assumes that all pages on the domain are allowed to be crawled.

4.4.10. Data Processing & Relevance Computation

One of the most important features of the crawling system is the processing of HTML documents and estimating their relevance for the given user input. In this chapter, the implementation of this processing is broken down in the order of execution.

When a HTML document is retrieved by a SessionWorker, it is parsed using Jsoup and all links are extracted from the document and then cached in a HashSet. Links are extracted with the help of the CSS selector `a[href]`. This selector matches all `<a>` tags that have a

“href” property, which stands for hypertext reference. Then, the SessionWorker initializes a new instance of a RelevanceCalculator. Here, the HTML document is cleaned from elements that are assumed not to be related to the main content of the page. Furthermore, elements that would not be displayed in a browser are removed, as well as media elements that do not contain text. The List of all these elements contains `<iframe>`, `<video>`, `<audio>`, `<canvas>`, `<embed>`, `<nav>`, `<header>`, `<footer>` elements, elements that have the property `role='navigation'`, elements that have a CSS class that contains the word “hidden”, and elements that have a style property that contains `display:none;`. These elements are removed from the DOM to make further traversals more efficient and to avoid irrelevant content to be part of further processing. For example, `<header>` and `<footer>` elements often contain lists or tables with navigation elements that are not related to the main content of the page. These lists would then be analyzed or sent to the ChatGPT API which would cost computing resources or waste ChatGPT tokens.

After the document is cleaned, the SessionWorker initializes two threads. One for computing the HTML Score and one for computing the Structured Data Score of the web page. The HTML Score is a combination of four sub scores which are computed in the following order:

1. **Entity Name Score** – This score is only calculated if the crawler focuses on a named entity and not just an entity type. First, all paragraphs and headings are extracted from the HTML document, by selecting all `<p>`, `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>` and `<h6>` elements. For all of these elements we check if the original query string or all of its tokens (separated by white space characters, in any order) are present in the element’s text, ignoring letter case. The ratio of elements containing the queried name to all selected elements is then cached. The same check is performed for the page’s `<title>` element, and the URL of the web page, resulting in either a value of one if the query is present, or zero otherwise. The final Entity Name Score is then returned as the arithmetic mean of the three values.
2. **Schema Property Score** – This score is calculated based on the schema.org properties of the queried entity type. It is computed by extracting the text of the document’s `<body>` element and counting all properties of the given schema.org type that are present in this text. A property is considered present in the text, if its humanized version (originally, schema.org properties are defined in camel case, “humanized” means that the property name is being separated by white space characters) or one of its synonyms matches the text, in a case insensitive manner. The final Schema Property Score is then returned as the ratio of present properties to all properties of the given type.
3. **Schema Type Score** – This score is calculated in a very similar way to the Entity Name Score. All headings and paragraph elements are being selected, and for each of those elements we check whether the humanized version of the entity type or all of its tokens are present in the element, ignoring letter case. The final score is then returned as the ratio of elements containing the entity type to all selected elements.
4. **ChatGPT Score** – This score is not calculated in place like the other HTML sub scores. It is based on the lists and tables of a web page. All `<table>`, `<ul>`,

 and <dl> elements are selected from the document, and their text is prepared for being passed to ChatGPT. This is done by replacing all newline and tab characters with white space characters (this avoids problems with the APIs JSON parser, that occurred during development), and by truncating the text to the maximum number of tokens that is allowed by the API. Then, the request is built and sent via Java's built-in HTTP client. The body of the request to OpenAI's API contains all necessary parameters and looks like this:

```
{
    "model": "gpt-3.5-turbo-0613",
    "messages": [
      {
        "role": "system",
        "content": "You are a useful tool for analyzing text. Your
                purpose is to evaluate whether a text snippet
              contains information about a music group. You return a
              score between 0 and 1 and NOTHING else. that score
              reflects your confidence that the supplied text
              contains information about a music group. If no
              information is present, you return 0."
      },
      {
        "role": "user",
        "content": "Origin: Liverpool, England
                Genres: Rock pop beat psychedelia
                Years active: 1960–1970
                Labels: Parlophone, Apple, Capitol"
      }
    ],
    "temperature": 0.2,
    "max_tokens": 100,
    "top_p": 1,
    "frequency_penalty": 0,
    "presence_penalty": 0
}
```

Listing 4.6.: Body of a ChatGPT request in JSON format

The purpose of the supplied parameters is explained in the following paragraph:

- **model** is the name of the model that should be used to process the supplied text. OpenAI provides many different language models for different purposes in multiple versions. "gpt-3.5-turbo-0613" was chosen because it was the most recent iteration of the available ChatGPT models at the time of development and performed the best in the initial trials that were conducted to settle on a model.
- **messages** contains the textual content that should be processed by the model. A "system" message is provided to give directions that introduce the model to its purpose for the request. The "user" message contains the extracted text from tables and lists on a web page.
- **temperature** controls randomness and a value between zero and two can be assigned. During first tests on the OpenAI playground[1], it showed that with the default value of one, the model would generate longer outputs, trying to explain the score it assigned to the data, or even just return an

[1] https://platform.openai.com/playground

explanation of the content, without assigning a numerical score to it. The parameter was then tweaked, until most of the time, the models answer would just contain the assigned score.

- **max_tokens** indicates the amount of tokens that the model is allowed to produce.
- **top_p** is used for nucleus sampling, which is a method of random sampling in the model's potential outputs. A value of 1.0 samples from all possible tokens, while values less than 1.0 make the model more focused. For this parameter, the default value of 1.0 was used, because no consistent impact could be noticed during initial tests.
- **frequency_penalty** and **presence_penalty** can be used to penalize or boost certain words or phrases from appearing in the model's output. Here, the neutral defaults were used because the output should always be a single number. The API responds with a JSON Object that looks like the example in Listing 4.7.:


```
{
    "id": "chatcmpl-7w8laacuVm2UllE2UIcxzZJL6poDx",
    "object": "chat.completion",
    "created": 1694091078,
    "model": "gpt-3.5-turbo-0613",
    "choices": [
        {
            "index": 0,
            "message": {
                "role": "assistant",
                "content": "1"
            },
            "finish_reason": "stop"
        }
    ],
    "usage": {
        "prompt_tokens": 105,
        "completion_tokens": 1,
        "total_tokens": 106
    }
}
```


Listing 4.7.: Example response of the ChatGPT API in JSON format

4.4.11. Error Handling

The system is designed to remain in an operational state even when encountering errors during the crawling process. The purpose of this is to ensure that the system can continue to function effectively, mitigating disruptions and minimizing downtime. Java provides several built-in mechanisms for error handling that are well-suited for creating robust and resilient applications.

In Java, exceptions are categorized into different types based on their characteristics and how they are handled. The most prominent kinds are listed below:

- **Checked Exceptions –** These are exceptions that a method is obligated to either catch or declare in its throws clause. They are checked at compile time which means the compiler will mandate to handle them, hence their name. Checked Exceptions are usually used to indicate errors that are outside the control of the program, e.g., in a method that tries to read a file from the file system and the file does not exist.
- **Unchecked Exceptions –** Also known as Runtime Exceptions, are exceptions that a method is not required to catch or specify. In most cases, this kind of exception reflects some kind of error in the program logic. For example, an Unchecked Exception would be thrown if a piece of code divides by zero or attempts to access a null object reference.
- **Errors –** These are serious issues that are usually not recoverable and are beyond the control of the program. For example, an OutOfMemoryError occurs, when the Java Virtual Machine is unable to allocate an object due to insufficient space in the Java heap. In a reasonable application, handling this kind of error is not advisable.

The crawling application is designed to catch and handle any checked and unchecked exceptions gracefully, enabling it to remain in an operational state when encountering problems. Especially in applications that are network-dependent and operate in a multithreaded manner, exceptions are likely to occur on a regular basis, which is why a reasonable error handling strategy is essential in such scenarios. Even more important is preventing errors from happening in the first place. The implementation of the system focuses on covering as many possible scenarios as possible, so that the crawler application can react to different conditions. For example, the functions exposed by the frontends REST-API follow a robust input validation strategy, preventing incorrect inputs and returning meaningful error messages when user input is incorrect.

Given that the application operates in a multithreaded environment, special care is taken to ensure thread safety. Concurrent data structures are used to manage shared resources like the URL frontier effectively, reducing the chance of race conditions that could result in unpredictable behavior or data corruption.

4.5. Frontend Application

As well as the crawler application, the frontend application utilizes the Spring framework. The application is based on a Model-View-Controller architecture, where the model layer of this architecture is outsourced to a shared library (named utils) which is also used by the crawler application. The frontend application only holds the controller and view components. The frontend provides a web interface that allows users to interact with the system in a user-friendly way; this interface was built using Thymeleaf. To allow other FactCheck++ services and applications to communicate with the system and utilize its features, a RESTful Application Programming Interface (REST-API) is provided.

4.5.1. Web Interface

The view component of the web interface is built using Thymeleaf[1], a modern server-side Java templating engine capable of processing and generating HTML, XML, JavaScript, and CSS. Thymeleaf was chosen because it integrates well with the Spring Framework, e.g., by offering the SpringStandard Dialect. It supports expressions for interactions with Spring's form tags and error messages. The web interface consists of three main pages:

1. **Dashboard –** The dashboard allows users to get an overview of the systems status. It contains a paginated table with all currently running or historic crawling sessions, including their query, number of resulting pages, starting date, and status. This table is sorted by the session's starting time, such that the newest sessions are always displayed on the first page. The dashboard also provides a health-check section, that allows users to quickly check the status of other system components. For each component (crawler, message broker and database), there is an icon that is either red or green, depending on the components status. From the dashboard, a user can navigate to the Details page for a specific crawling session, as well as to the New Session page to configure new crawling sessions.

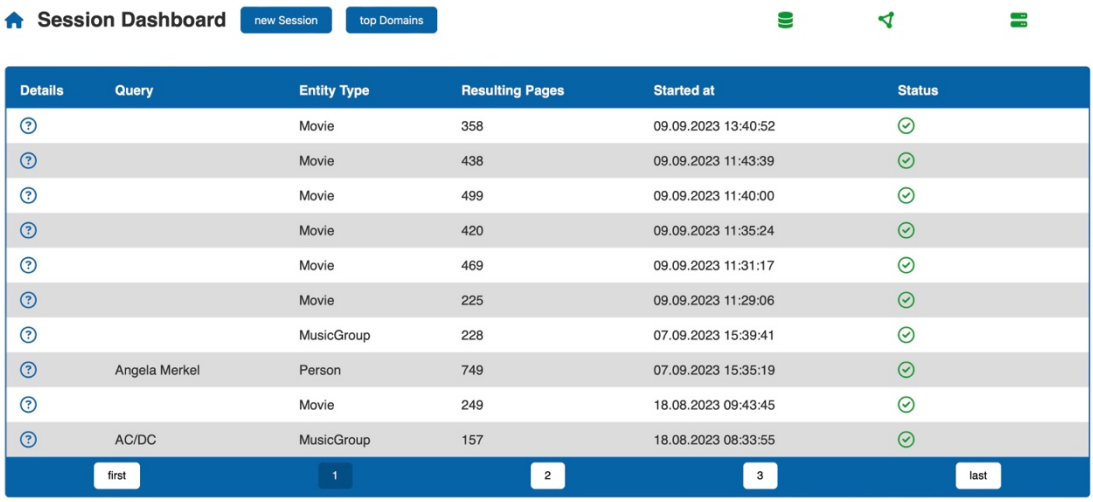


Figure 4.6.: Screenshot of the session dashboard page

2. **Details –** The details page contains the complete data of a crawling session; all user supplied input parameters, the database ID of the session, the number of URLs that have been visited by the crawler, the number of processed URLs (processed URLs is a superset of visited URLs; the crawler might process an URL and decide not to visit it, e.g., because it is not allowed by the domains Robots Exclusion Protocol), the number of failed requests, and the number of ChatGPT tokens used during the session (including their estimated cost). The details page also shows a table with all URLs resulting from the crawling session, together with their calculated scores (HTML Score and its sub-scores, Structured Data Score, and the maximum of

[1] https://www.thymeleaf.org

HTML- and Structured Data Score). By default, this table is sorted by the maximum Score, such that the highest-ranking URLs are on top. The user also has the possibility to sort by HTML Score or Structured Data Score.

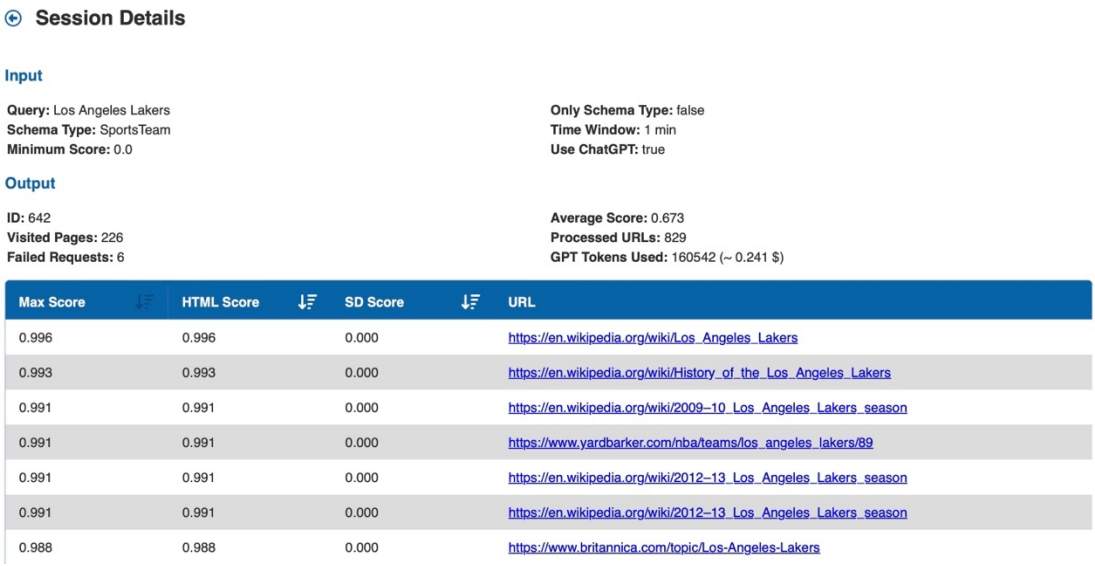


Figure 4.7.: Screenshot of the session details page

3. **New Session** – On this page, the user can configure and trigger a new crawling session. It contains a form with fields for all the possible parameters and a confirm button.

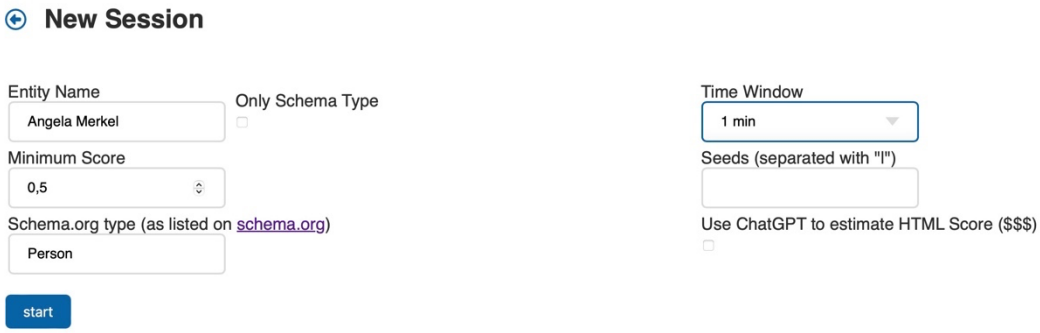


Figure 4.8.: Screenshot of the new session page

To serve these pages to the client, a dedicated Web Controller was implemented using the Spring framework, which exposes an endpoint that serves server-side rendered HTML. Its functions return the page templates populated with data based on query parameters. Tables 4.1 – 4.8 show those endpoints, their functions, and parameters.

Title	Get Dashboard
URL	/
Method	GET
Parameters	page: Integer
Success	200 – Dashboard Template populated with paginated session information
Error	500 – The Server could not return the requested page

Table 4.1.: Details of the get dashboard function in the web endpoint

Title	Reload Dashboard (without reloading the whole page)
URL	/load_dashboard
Method	GET
Parameters	page: Integer
Success	200 – Dashboard Template populated with paginated session information
Error	500 – The Server could not return the requested page

Table 4.2.: Details of the reload dashboard function in the web endpoint

Title	Get Session Details
URL	/details
Method	GET
Parameters	idSession: Integer (required) sortField: String (required)
Success	200 – Details Template populated with detailed session information
Error	500 – The Server could not return the requested page

Table 4.3.: Details of the session details function in the web endpoint

Title	Get New Session Template
URL	/new_session
Method	GET
Parameters	
Success	200 – New Session Form Template
Error	500 – The Server could not return the requested page

Table 4.4.: Details of the new session function in the web endpoint

Title	Trigger new Session
URL	/new_session
Method	POST
Parameters	Session: Session Model
Success	200 – Session created
Error	500 – The Session could not be created

Table 4.5.: Details of the trigger new session function in the web endpoint

Title	Cancel Session
URL	/
Method	POST
Parameters	idSession: Integer (required)
Success	200 – Session cancelled
Error	500 – The Session could not be cancelled

Table 4.6.: Details of the cancel session function in the web endpoint

Title	Receive Session updates
URL	/events
Method	GET
Parameters	
Success	200 – SseEmitter (text/event-stream)
Error	500 – SseEmitter could not be created

Table 4.7.: Details of the receive session updates function in the web endpoint

Title	Receive Notifications from Backend
URL	/notifications
Method	GET
Parameters	
Success	200 – SseEmitter (text/event-stream)
Error	500 – SseEmitter could not be created

Table 4.8.: Details of the receive notifications function in the web endpoint

The pages are based on Thymeleaf templates, which are inserted into a master template. This means, when a user visits the web interface in a browser, the server will return only the master template and replace the page templates based on user input. This prevents the browser from reloading the entire website when a user goes to another page.

To keep the Web Interface updated during crawling sessions, Server-Sent Events (SSE) technology was employed. SSE is a server-side push technology that enables a browser to receive automatic updates from a server via HTTP connection. This feature was designed to be used over HTTP (or HTTPS), which makes it a simple yet robust solution for establishing a unidirectional communication channel from the server to the client. Once a user visits the Web Interface in a browser, the client establishes an SSE connection via the JavaScript EventSource API to the /notifications Endpoint. While a crawling session is running, the crawler will notify the Frontend every five seconds via a dedicated RabbitMQ Queue that new results are available. When the Frontend receives a message via this Queue, it sends an update to connected clients via the SseEmitter the clients subscribed to. If the browser would reload the entire website when a user goes to another page, the SSE connections to the server would be destroyed, causing the frontend to miss notifications or updates from the server. This approach has some drawbacks; for example, the "back" button of the browser will not work as intended for the web frontend. For this reason, the web interface has its own dedicated "back" button which redirects the user to the Dashboard page. Furthermore, the URL displayed in the browser window will not reflect the site hierarchy of the application, as there is only one URL for the master – or index – template.

4.5.2. REST-API

The REST-API plays an important role in the architecture of the crawling system, offering a programmatic interface for orchestrating crawling sessions. This API exposes an endpoint with various functions that cater to different user needs and enable seamless integration with other systems or tools in the FactCheck++ framework. The API has been designed to be intuitive, adhering to RESTful principles to ensure ease of use. Tables 4.9 – 4.13 show the details of the provided functions:

Title	Get all structured data triples from a given URL
URL	/rest/triples?url={url}
Method	GET
Parameters	url: String (required)
Success	200 – List of triples in turtle format
Error	500 – The Server failed to extract triples from the given URL 400 – No URL was provided

Table 4.9.: Details of the get triples function in the rest endpoint

The function displayed in Table 4.9 was initially designed for testing the structured data extraction capabilities of the Apache Any23 library. It expects an URL string as a parameter. Upon receiving a request, the system will employ the JsoupUtil component to download the web page referenced by the supplied URL and extract all structured data triples on those pages using the Any23 library. The triples are then formatted in Turtle format and returned in the response body. The function was kept in the endpoint because

there might be other FactCheck++ services that could use a simple structured data extraction tool like this.

Title	Get data of a crawling session
URL	/rest/sessions/{id}
Method	GET
Parameters	id: Integer (required)
Success	200 – Serialized JSON object containing input and output of the requested session
Error	500 – The Server failed to return the requested session data 400 – No ID was provided 400 – Session with given ID does not exist

Table 4.10.: Details of the get session function in the rest endpoint

The function displayed in Table 4.10 provides results of a previous or currently running crawling session. The function takes the numeric identifier of a session as a path parameter. The data returned by the function contains all information that would be displayed on the results page of the web interface, including all input parameters, session metadata like estimated cost or number of visited pages, and the details on each crawled page. A typical response is shown in Listing 4.8.:


```
{
    "input": {
        "query": null,
        "schema_type": "MusicGroup",
        "time_window": "PT1M",
        "min_score": 0.0,
        "use_gpt": true,
        "only_schema_type": true
    },
    "output": {
        "id": 649,
        "visited_pages": 228,
        "failed_request": 0,
        "processed_urls": 228,
        "gpt_tokens_used": 151437,
        "gpt_price": 0.2271555,
        "results": [
            {
                "html_sub_scores": {
                    "chat_gpt_score": 1.0,
                    "entity_name_score": 0.0,
                    "entity_type_score": 0.735294117647059,
                    "schema_properties_score": 0.297619047619048
                },
                "url": "https://en.wikipedia.org/wiki/ the_Basic_Cable_Band",
                "html_score": 0.999475612142043,
                "structured_data_score": 0.0
            },
            ...
        ]
    }
}
```


Listing 4.8.: Example response of the REST-API for a single crawling session in JSON format

Title	Get data of a running crawling session as a stream
URL	/rest/sessions/{id}/stream
Method	GET
Parameters	id: Integer (required)
Success	A real-time stream of results
Error	500 – The Server failed to return the requested session data 400 – No ID was provided 400 – Session with given ID does not exist

Table 4.11.: Details of the get stream function in the rest endpoint

The function displayed in Table 4.11 serves the same purpose of the previous function but is intended to be used for running crawling sessions. The function takes the numeric identifier of a session as a path parameter and returns a stream (MIME type text/event-stream) of JSON objects containing details on crawled pages in real time. The controller which implements the rest endpoint holds a map of all active SSE emitters that handle the event streams. Upon receiving a stream request, the controller initializes a new SSE emitter instance. The controller also implements a listener for a dedicated RabbitMQ queue that receives a message from the crawler application every time a new web page was processed. This listener then checks whether an SSE emitter exists for the session id encoded in the message received from the crawler application. If so, the listener passes the received message to the corresponding SSE emitter. A single message returned in the stream looks like the example in Listing 4.9.:

```
{
    "html_sub_scores": {
        "chat_gpt_score": 1.0,
        "entity_name_score": 0.0,
        "entity_type_score": 0.597560975609756,
        "schema_properties_score": 0.404761904761905
    },
    "url": "https://en.wikipedia.org/wiki/Incubus_(band)",
    "html_score": 0.999357062055166,
    "structured_data_score": 0.0
}
```

Listing 4.9.: Example element of the event stream returned for a running crawling session in JSON format

Title	Cancel a running crawling session
URL	/rest/sessions/{id}/cancel
Method	GET
Parameters	id: Integer (required)
Success	200 – Session is cancelled
Error	500 – The Server failed to cancel the session 400 – No ID was provided 400 – The session is already finished

Table 4.12.: Details of the cancel session function in the rest endpoint

The function displayed in Table 4.12 can be used to cancel a running session. It also takes the Id of a crawling sessions as a path parameter. Upon receiving a cancellation request, the function checks whether the requested session is existing and running. If so, the controller updates the status of the session to "cancelling" and a RabbitMQ message is sent to the Scheduler component of the crawling application containing the session Id. The Scheduler then is responsible for terminating the crawling session.

Title	Start a new crawling session
URL	/rest/sessions
Method	POST

Parameters	- return_stream: Boolean (optional),
	- Session parameters as serialized JSON object in the request body, e.g.
	<pre>{
	"query": "Microsoft",
	"only_schema_type": false,
	"min_score": 0.7,
	"time_window": "PT1M",
	"schema_type": "Organization",
	"use_gpt": false,
	"seeds": []
	}</pre>

Success	200 – Session created OR a real-time stream of results, depending on the value of return_stream
Error	500 – The Server failed to start the session 400 – Parameter "schema_type" is missing or invalid 400 – Parameter "query" is missing 400 – Parameter "time_window" is missing or invalid 400 – Parameter "only_schema_type" is missing 400 – Parameter "min_score" is missing or invalid

Table 4.13.: Details of the trigger session function in the rest endpoint

The function displayed in Table 4.13 can be used to trigger a new crawling session. It expects a JSON object containing the input parameters in the request body, as shown in the table above. Upon receiving a request, the function checks whether all required parameters are present and in the desired format. If so, a session entry is inserted to the database, with status "pending". Then, a RabbitMQ message is sent to the Scheduler component of the crawling application via a dedicated queue. The Scheduler is then responsible for starting the crawling session by instantiating a new SessionMaster instance. By default, the function returns the Id of the newly created session in the response body. The function is also capable of directly returning a stream of crawling results, just like the function that is responsible for returning session details. This can be achieved by setting the boolean parameter "return_stream" to "true".

4.6. Communication

The communication between the frontend and backend systems is facilitated by RabbitMQ, an open-source message broker that operates on the Advanced Message Queuing Protocol (AMQP). This setup offers several notable benefits:

1. **Decoupling –** RabbitMQ allows the frontend and the crawler application to operate independently. This separation ensures that changes or issues in one do not directly affect the other application.
2. **Stability –** Due to the separation facilitated by RabbitMQ, the system can achieve greater stability. If the crawler application encounters an issue, the frontend remains functional, providing consistent user interaction.
3. **Scalability –** The use of a message broker promotes system expansion. The system can be scaled horizontally by deploying more crawler instances that access the same message queue. This way, multiple crawling sessions could be performed on dedicated nodes concurrently.
4. **Load Balancing –** In a scenario with multiple crawler instances operating on different nodes, RabbitMQ could act as a load balancer, distributing new crawling sessions evenly across nodes, ensuring no single node is overloaded. This distribution promotes efficient resource use and consistent performance.
5. **Deployment Flexibility –** With RabbitMQ, the system can adapt to different deployment strategies, be it cloud-based, hybrid, or on-premises. This adaptability ensures the system can meet varying operational needs. In this case, RabbitMQ is running in a Docker container, like all other system components, which makes deploying the system a simple task, as explained in 4.7. Deployment

In summary, using RabbitMQ for communication between the frontend and backend offers a stable, scalable, and flexible system, well suited for the requirements of the system. In total, the system uses five queues serving dedicated purposes:

1. **CrawlerUpdateQueue –** This queue is used to inform the web frontend controller that new pages have been crawled, a SessionMaster sends ping messages to this queue in five second intervals while the crawling process is running.
2. **HealthQueue –** This queue is used to test the communication between the frontend application and the crawler application. On startup, the frontend application sends a ping to the HealthQueue, which should be returned by the crawler application.
3. **RestStreamQueue –** This queue is used to obtain crawling results in real time from the crawler. SessionWorkers send details of each crawled page to this queue, which is then read from the REST controller in the frontend application. When no client requested a stream for a given session, the messages are discarded.

4. **TerminateSessionQueue** – Using this queue, the frontend application notifies the crawler when a user requested a session to be terminated. The Scheduler component of the crawler, which is responsible for managing sessions, listens to this queue and terminates sessions based on the session Id that is expected to be contained in the messages.
5. **TriggerSessionQueue** – This queue is similar to the previous mentioned queue but serves the purpose of informing the crawler that a new session should be started.

4.7. Deployment

The entire System can be containerized and deployed with the use of Docker Compose. Docker Compose facilitates the definition and orchestration of multi-container Docker applications, simplifying the otherwise complex process of managing different services that make up the system. In this case, the system consists of multiple services: the frontend application, the crawler application, a RabbitMQ instance for message queuing, and a PostgreSQL database instance. Each of these services is isolated in its own Docker container, making use of a dedicated Dockerfile that contains the specific build instructions and configurations. Dockerfiles are written in a proprietary format and contain all instructions a user could call on the command line to build a Docker image. For example, the Dockerfile of the crawler application looks like this:

```
FROM amazoncorretto:17-alpine-jdk
   MAINTAINER plonusl94
   ADD build/libs/dowork-0.0.1-SNAPSHOT.jar /opt/lib/dowork.jar
   ENTRYPOINT ["java", "-jar","/opt/lib/dowork.jar"]
```

Listing 4.10.: Dockerfile for the crawler application

The script shown in Listing 4.10 creates a new Docker image based on an existing image with an Alpine Linux distribution and JDK 17 (Java Development Kit) installed. It then copies the pre-built .jar file of the local machine to the image and specifies the command to be executed when a container of this image is started. This command simply executes the provided java application.

Docker Compose allows for seamless integration of these separate containers. By combining them into a single unit using a docker-compose.yml configuration file, the system can be deployed as a cohesive, isolated unit across various environments. Docker Compose files are written in YAML (YAML Ain't Markup Language) and are used to define and manage the entire lifecycle of an application's components, including starting, stopping, building, and scaling services, with simple command-line instructions. This flexibility has real-world implications for deployment, particularly in cloud scenarios. One of the key advantages of containerization is environmental consistency. Containers encapsulate not just the application itself, but also its runtime environment. This encapsulation ensures that the application behaves uniformly, irrespective of where the container is deployed. This is especially useful for debugging and testing purposes, as it mitigates the old problem of "but it works on my machine", where code behaves differently on different machines due to environmental inconsistencies. Isolation is another significant

benefit. Since each container runs in its own isolated environment, there is little to no risk of interference between multiple applications running on the same host. This isolation is crucial for system stability and security, as it minimizes the chances of one application's processes affecting another's. Additionally, the containerized approach aligns well with DevOps practices and Continuous Integration/Continuous Deployment (CI/CD) pipelines. With all dependencies bundled within each container, migration across different stages of development becomes far more straightforward. This is particularly useful for the FactCheck++ framework, which operates on Microsoft's Azure platform. The containerized system can be easily migrated to Azure, or any other cloud provider, without requiring any alteration to the application code or its dependencies, thus streamlining the deployment process.

5. Experimental Evaluation

In the previous chapters, the design and implementation of the relevance metrics for web pages have been covered. In this chapter, the meaningfulness of those metrics will be discussed, as well as the methods that were used to evaluate the metrics.

5.1. Data Collection

To evaluate whether the developed relevance metric represents the relevance of a web page given the user's query, four crawling sessions have been conducted. The results of those sessions were logged into separate spreadsheets. For each resulting URL, all calculated scores (Entity Type Score, Entity Property Score, Entity Name Score, ChatGPT Score, HTML Score, Structured Data Score) were recorded, as well as all data that is required to calculate these scores.

Runtime	**Entity Name**	**Entity Type**	**# of visited URLs**
5 min	Angela Merkel	Person	1431
1 min	Los Angeles Lakers	SportsTeam	207
1 min		MusicGroup	204
1 min		Person	292
			2134

Table 5.1.: Summary of crawling sessions used to evaluate the system

In the first session, the crawler was configured to run for five minutes and to focus on the Entity "Angela Merkel" of type "Person". During the session, 1431 pages were visited and evaluated by the crawler. This session was conducted before the URL Frontier's priority and sorting mechanisms were implemented, which means that during this session, the crawler followed a breadth-first-search strategy. This allows to compare the two strategies in terms of harvest rate. The other three sessions were conducted using the best-first strategy as explained in section 4.4.5. The method used to calculate the relevance scores is the same across all sessions.

The remaining three sessions were configured to run for 1 minute, and the queries were chosen to cover a variety of topics, including specific entities and entity types.

5.2. Method and Results

To evaluate whether the scores assigned by the crawler actually represent the relevance of crawled pages, each of the 2134 resulting pages were visited in a web browser (Safari 16.4) and assigned a binary rating whether the page was relevant to the given query. If the page contained at least one fact about the queried entity, the page would be marked as relevant; if there was no statement about the queried entity, the page would be considered non-relevant. To get a first impression of the data, the results of the first session underwent a thorough investigation. First, the correlation between the assigned HTML Score and the actual relevance was calculated. Since HTML Score is a continuous variable and actual

relevance is a dichotomous variable, the Point-Biserial Correlation Coefficient was calculated, which is mathematically equivalent to the Pearson Correlation. The results show a high positive correlation between the two variables, with $r(1399) = .72$, $p < .001$.

Besides the correlation, some interesting observations have been made while investigating the data. The calculated HTML Scores, even the highest ones, tend to be relatively low. The first dataset had an average HTML Score of .375 for relevant pages and .229 for non-relevant pages. The maximum HTML Score of this dataset was .629. That could be explained by the way the sub scores of the overall HTML Score are calculated. For example, the Entity Type Score is denoted by the ratio of paragraphs and headings on a page containing the entity type to all paragraphs and headings on a page. This ratio is influenced by the amount of content on a page. E.g., a page that has 20 paragraphs and headings, from which five contain the queried entity type would score higher than a page with 100 paragraphs and headings, from which ten contain the queried entity type, although the absolute number of "relevant" elements would be higher on the second page. The same holds true for the Entity Name score, as it also incorporates such a ratio, but the effect is not as strong because Entity Name Score also includes whether the URL and title of the given page contain the entity name.

The second observation was, that it was hard to distinguish a relevant page from a non-relevant page only by looking at its HTML Score. An average HTML Score of .375 for relevant pages does not convey the intended feedback to the user.

The insights gained from those observations were used to adjust the HTML Score calculation method. The most obvious strategy to make differences in the scores more apparent to users of the system would be to use a min-max normalization. This would help to spread the scores more evenly in the range of zero to one. There are two problems with this method, though. First, it would require the system to finish crawling before being able to return the final scores of the crawled pages, because the minimum and maximum scores are only available when all pages have been visited. As the system is designed to return results in real-time, normalization has to be done "in-place", without relying on other results. The second problem with min-max normalization is, that it destroys comparability and possibly the reliability of scores. For example, if min-max normalization was applied based on the results of a single crawling session, a session that only sees non-relevant pages would still return high scores for those pages.

To discriminate the lower scores from the higher scores and reach higher coverage of the spectrum from zero to one, a sigmoid function was introduced to the final HTML Score. A sigmoid function is a mathematical function which outputs a value between zero and one and is defined as:

$$f(x) = \frac{1}{1 + e^{-x}}$$

These functions are commonly used in the field of Artificial Intelligence, specifically for training Neural Networks and for binary classification tasks. The basic sigmoid function was adapted according to the before mentioned observations, in this case x would be the original HTML Score:

$$f(x) = \frac{1}{1 + e^{-20\,(x - 0.3)}}$$

The threshold – or inflection point – of the modified function is moved to f(0.3). This value was chosen based on the difference between the mean HTML Score of relevant pages (0.375) and the mean HTML Score of non-relevant pages (0.229). This way, scores below this threshold will be penalized and scores above the threshold will receive a boost:

$$thresh = 0.375 - \frac{0.375 - 0.229}{2} \approx 0.3$$

The steepness of the original sigmoid function was also modified to ensure that its output is nearly equivalent to its input as the input approaches zero or one. A plot of the resulting sigmoid function is shown in Figure 5.1.

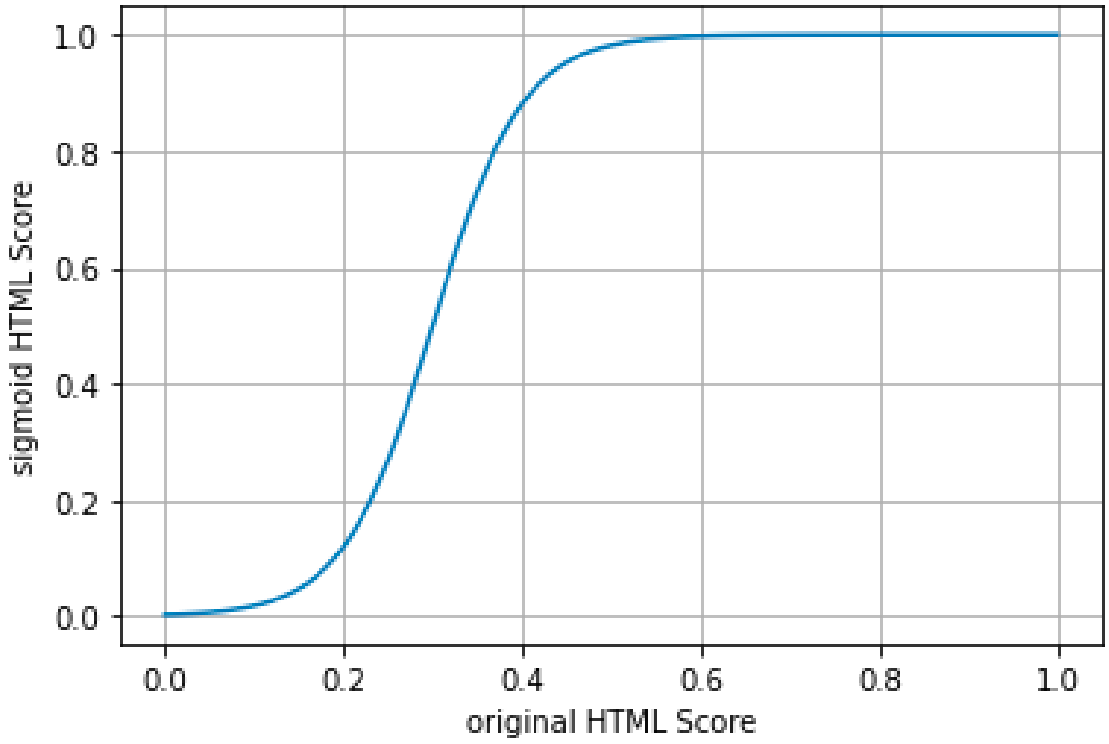


Figure 5.1.: Plot of the sigmoid function

To validate the choice of the sigmoid function and to ensure that it is applicable not only for the query used in the first session, the sigmoid function was applied to all HTML Scores of the three remaining datasets. Then, the Point-Biserial Correlation between the actual relevance and the final HTML Score was calculated for those datasets, as displayed in Table 5.2.

Entity Name	Entity Type	Correlation original HTML Score – Relevance	Correlation sigmoid HTML Score - Relevance
Angela Merkel	Person	r(1429) = .72, p < .001	r(1429) = .73, p < .001
Los Angeles Lakers	SportsTeam	r(205) = .88, p < .001	r(205) = .93, p < .001
	MusicGroup	r(202) = .73, p < .001	r(202) = .77, p < .001
	Person	r(290) = .72, p < .001	r(290) = .72, p < .001

Table 5.2.: Correlation between HTML Score and actual relevance before and after applying the sigmoid function

The results show that applying sigmoid function to the HTML Score does increase the correlation between the score and the actual relevance. The intended effect of discriminating scores for relevant pages from scores for non-relevant pages was achieved. Table 5.3 shows the mean of those scores before and after applying the sigmoid function.

Entity Name	Entity Type	Mean original Score (Median)	Mean sigmoid Score (Median)
Angela Merkel	Person	Relevant: **0.37** (0.39) Non-relevant: **0.23** (0.22)	Relevant: **0.65** (0.85) Non-relevant: **0.23** (0.15)
Los Angeles Lakers	SportsTeam	Relevant: **0.45** (0.47) Non-relevant: **0.12** (0.12)	Relevant: **0.92** (0.97) Non-relevant: **0.12** (0.03)
	MusicGroup	Relevant: **0.49** (0.51) Non-relevant: **0.06** (0.04)	Relevant: **0.92** (0.99) Non-relevant: **0.05** (0.01)
	Person	Relevant: **0.43** (0.45) Non-relevant: **0.10** (0.10)	Relevant: **0.85** (0.95) Non-relevant: **0.05 (**0.02)

Table 5.3.: Means of the HTML Score before and after applying the sigmoid function

In summary, the results of this experimental evaluation show that the crawling system can effectively differentiate between relevant and non-relevant pages in most cases. An aspect of the collected data that should be highlighted in this context is the fact that the system occasionally assigns scores that do not reflect the actual relevance of a web page. The data show that the system tends to produce false negatives in those cases, which means, that low scores are assigned to pages that are relevant. The following box plots show the distribution of those outliers for two of the test sessions.

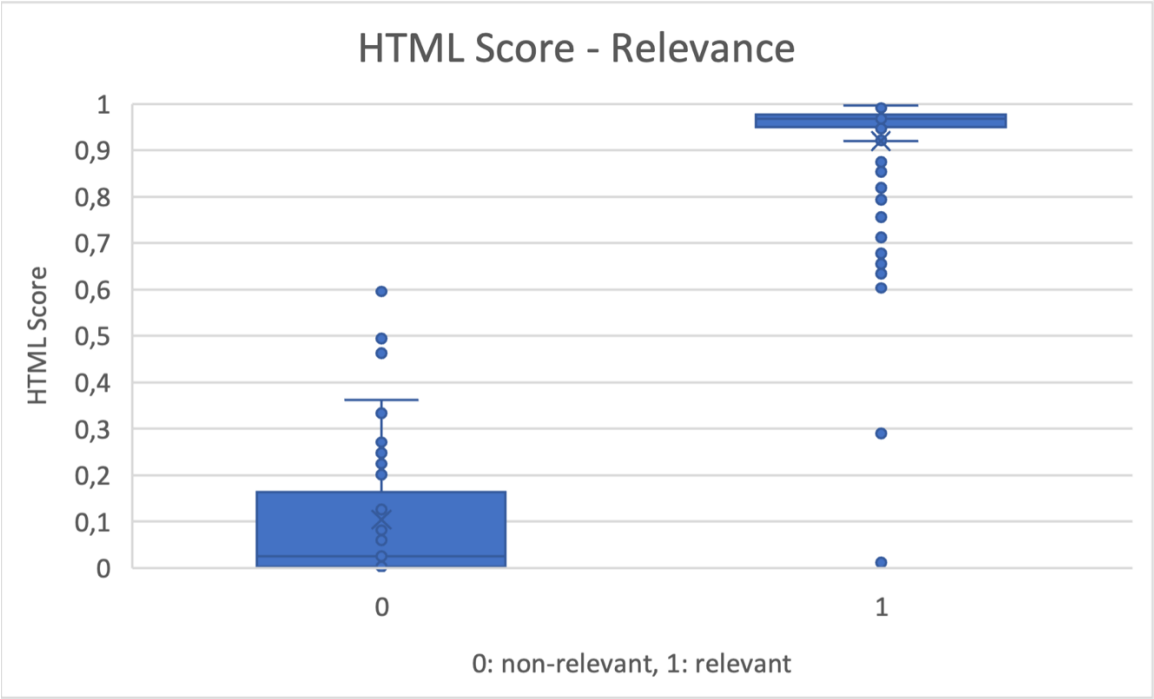


Figure 5.2.: Box plot of the HTML Score assigned to relevant and non-relevant pages for the topic Los Angeles Lakers

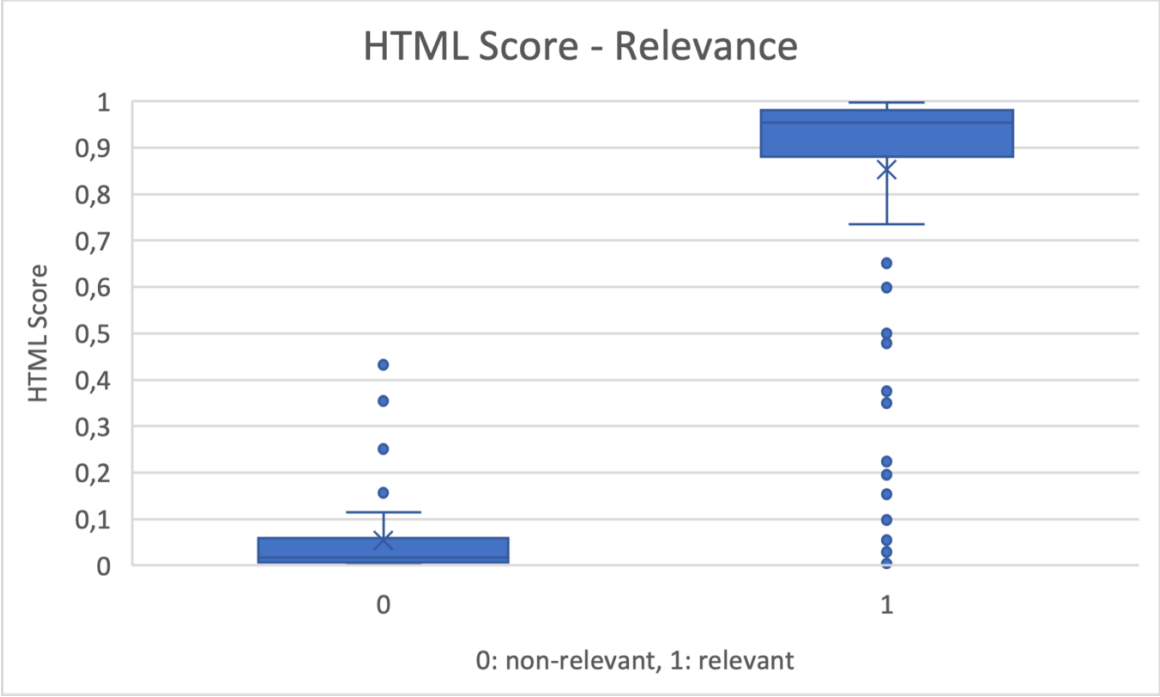


Figure 5.3.: Box plot of the HTML Score assigned to relevant and non-relevant pages for the topic Person

As mentioned in section 5.1., the collected data allows to compare the harvest rate of the breadth-first and best-first crawling strategies. During the first session, which used the

breadth-first strategy, the crawler visited 1431 web pages, of which 199 are considered relevant for the given query, which equates to 13.9 %. During the other three sessions, the crawler visited a total of 703 web pages, of which 599 are considered relevant: 85.2%. Whether these numbers reflect the actual harvest rate, depends on the definition of the term harvest rate. The universally accepted definition of harvest rate is the ratio of relevant web pages that are retrieved by all retrieved web pages [40], [41]. Using this definition, the aforementioned numbers do reflect the actual harvest rate. But as the proposed crawling system assigns relevance scores to visited pages, it could be argued, that the harvest rate for this crawler should be defined as the ratio of relevant pages that are rated with a high relevance score by all retrieved pages. However, using this definition to calculate the harvest rate, two variables are mixed: the priority sorting mechanism and the relevance computation method. This makes drawing conclusions from the following numbers not applicable, especially because the definition of "high relevance score" also depends on the use case. Keeping this in mind, if we use the adapted definition of harvest rate and define a HTML score greater than 0.5 as high, the breadth-first crawling strategy yields a harvest rate of 8.7% and the best-first strategy comes at a harvest rate of 79.9%.

5.3. Robustness and Performance Test

To test whether the system is capable of crawling for extended periods of time, without running into performance issues or other unforeseen problems, a long-running crawling session was performed. This session was configured to run for 24 hours crawling for the topic "Angela Merkel" with the entity type "Person". No initial seed URLs were provided. The use of the ChatGPT API was deactivated because of monetary concerns, and the quality of the resulting scores was not of interest in this test. With ChatGPT disabled, it was also possible to gain insights to the performance of the system without the rate limiting of third-party APIs.

During the session, the crawling system processed 632,493 unique URLs which equates to 439.23 URLs per minute on average. The system handled all occurring Exceptions gracefully without crashing or producing memory issues.

6. Discussion

In this chapter, some design decisions, the reasoning behind those, and characteristics of the project are being discussed.

6.1. Cost of ChatGPT

The use of OpenAI's ChatGPT API introduces cost to the operation of the proposed system. The API is a paid service, and the price is based on the number of tokens that are submitted to the API and the number of tokens that are generated by ChatGPT. At the time of development, OpenAI charges $0.0015 per 1000 input tokens and $0.002 per 1000 tokens generated by the AI. A token is approximately four characters long or 0.75 words for English text, as OpenAI states in the FAQ section of their website [42]. The crawling system keeps track of the cost generated by using ChatGPT for each session. The API returns the number of used tokens for each request, so the crawler can sum up those values during a crawling session and calculate the total cost of the session. The system uses the following formula to calculate session costs in US Dollars:

$$cost = \frac{\mathrm{N}_{tokens}}{1000} \times 0.0015$$

This method neglects the higher cost of output tokens. This is certainly not correct, but for estimating the cost of a session it is a sufficient approximation, as we expect only 1 output token per request.

The exact cost per session varies slightly based on the amount of web pages the crawler visits per minute and on the structure of those pages. During the last test sessions of the system, the crawler visited roughly 250 web pages per minute, using around 150,000 ChatGPT tokens. Based on these numbers, processing a single web page introduces a cost of $0.0009 and crawling for one minute costs $0.225 on average. On a larger scale, running multiple instances of the crawler for extended amounts of time, this could generate significant cost. However, the ChatGPT API allows for configuring monthly usage limits with e-mail notifications, which can help keeping track of costs and avoid spending unwanted amounts of money. Furthermore, the system can be configured to skip the ChatGPT processing for a session, which is helpful for tests or exploratory crawling sessions that do not require this external service.

During development, OpenAI reduced the cost of ChatGPT input tokens by 25% [43]. If OpenAI continues with these kinds of updates to their pricing policy, the crawling system would profit from that, because the main cost factor are input tokens. OpenAI also announced a researcher access program, that allows researchers to claim API credits for their studies [44].

6.2. Relevance Metric Design

The design of the relevance metrics was an iterative process, which included many ideas that were tested and discarded during the systems design and implementation. For example, the initial design did not include the separation of HTML Score and Structured Data Score. The idea was to return a single score which would be the arithmetic mean of those scores; The reasoning behind this was to enable users to quickly assess page relevance by looking at a single number. During development, it became evident that this approach would effectively hide information from users and made it hard to comprehend the reasoning behind assigned scores. Because of this, the decision was made to return two distinct scores for each visited web page. One score that reflects the relevance of the structured data on a page, and one that reflects the relevance of the HTML encoded content of the web page. This makes it easier for users to understand the assigned scores and should help other information extraction systems to decide which kind of information to extract. Furthermore, all sub scores of the HTML Score are returned for each page, serving the transparency of the relevance calculation.

6.3. Extensibility of the System

The mechanisms the crawler application uses for computing relevance is fixed, as well as some other configurations, like the version of the schema.org ontology. For example, altering the weights of the sub scores of the HTML Score is not possible without changing the source code, and HTML tags that are considered in these scores are predefined in the source code. These configurations might be subject to changes in the future, for example when new HTML elements are introduced to the standard, or when expert users feel the need to change the weights of sub scores. A production ready system should provide the possibility to configure such mechanisms, future-proofing the entire system by making it adaptable to changes that cannot be foreseen during development. These kinds of extensibility could be provided either by implementing a *Global Settings* page for the systems web interface or using environment variables. Usability would not benefit from the latter, as changing environment variables is nearly as intricate as changing the source code directly but would spare the effort to extend the systems data model and database setup. As mentioned in section 7.5, a *Global Settings* page is already planned for future development iterations, so this could be the appropriate place for the extensions proposed in this section.

6.4. Performance of the System

The system's performance is heavily dependent on network characteristics and the performance of the third-party APIs the crawler employs. While network issues are not directly related to the implementation of the system, the use of third-party APIs is an intentional design choice and drawbacks resulting from this choice should be justified. The largest impact on performance comes from the rate-limits that are installed on the OpenAI API. As the API only allows for 90.000 tokens to be processed per minute, the systems SessionWorker threads are forced to pause execution from time to time, to prevent error

responses from the API. This causes a significant drop in the systems crawl-rate. As stated earlier, in a crawling session that makes use of the ChatGPT API, the crawler visits 250 web pages per minute on average. When skipping those API calls, the system visits around 500 web pages per minute, which equates to a 100% performance boost, with the cost of lower precision in relevance estimation. As the system is designed to allow users to configure if the ChatGPT should be employed in a crawling session, this characteristic could be exploited for adapting the system to different use cases. Crawling sessions that do not employ ChatGPT could be used for initial explorations to quickly gather an initial set of URLs that the system can use as seeds for future crawling sessions.

7. Future Work

This chapter focuses on the possible extensions and improvements of the crawling system, that should be implemented in future development of the project.

7.1. Use of Large Language Models

As the current use of ChatGPT to estimate page relevance shows promising results, this approach could be developed further. Since the development in the field of LLMs has gained attention by many developers, including big corporations like Google and Meta, but also by the open-source community, it is to be assumed, that high quality, open-source, and self-hostable LLMs will be available in the near future. For example, Meta released Llama 2 in July 2023 [45], an open-source competitor to ChatGPT, which cannot keep up to ChatGPT in terms of parameter size, but has the advantage of being available for free. Also, other open-source projects like GPT4All [1] experience high levels of attention and contribution. A self-hosted, open-source LLM could improve the crawling system's performance in multiple ways:

- **Fine-Tuning –** Models could be fine-tuned to the specific use case of the crawling system, potentially increasing precision of results.
- **Cost Efficiency –** By providing own infrastructure to host a LLM, the cost for crawling sessions could be reduced, compared to using the paid ChatGPT API.
- **Performance –** The rate limits of the ChatGPT API act as a performance bottleneck in the crawling system. By avoiding these rate limits, the crawler would be able to process more pages per minute.

During the development of the system, OpenAI released several updates to its API, which contain features that could possibly be useful for the crawler. For example, a new version of the ChatGPT model, GPT-4, became available to the public [46], and OpenAI announced the possibility to fine tune their models to improve steerability and reliability [43]. Employing these new features in the crawling system could possibly improve performance and precision of the score calculation.

7.2. Scaling the System

The proof-of-concept implementation of this project is designed to run on a single node. However, the possibility of scaling the system horizontally was always part of the decisions made during development. To distribute system load over multiple nodes, two scenarios were considered:

[1] https://gpt4all.io/index.html

- **Inter-Session scaling** – In this scenario, individual crawling sessions are performed on a single node. Multiple crawler instances would be deployed, such that concurrent crawling sessions could be performed on dedicated nodes. The current implementation of the system supports this approach. By deploying multiple instances of the crawler application which all connect to the same RabbitMQ and Frontend instances, this form of scaling could be achieved with minimal configuration effort. To increase independence between crawler instances, dedicated API keys should be provided. Otherwise, rate limits of those APIs could become a bottleneck for the whole system, and the purpose of scaling the application would be diminished.
- **Intra-Session scaling** – Another way of scaling the application would be to distribute the load of individual crawling sessions between multiple instances of the crawling application. For this purpose, the URL frontier could be outsourced to RabbitMQ, such that each crawling session has a dedicated queue. AMQP supports priority queues, so it is possible to keep all URL priority calculation logic and just outsource the PriorityBlockingQueue implementation currently used in the crawler to a RabbitMQ priority queue.

7.3. Language Support

As mentioned in 3.5.2. HTML Score, the crawling system should be able to deal with different languages that occur on the web. To achieve this, the existing schema.org ontology can be extended with different languages. As all entity types, properties, and their synonyms are stored in database tables, the most efficient approach to this would be to add a column to these tables for each language that should be supported. For example, the *schema_properties* table has a column *synonyms,* that holds the different expressions for a property in English language. By adding a column *synonyms_de*, which would hold those expressions in German language, the crawler could look up the expressions in the currently needed language by checking the web page's language attribute, usually defined in its `<html>` tag.

7.4. Logging

In its current state, the system logs messages exclusively to the crawler application's console output. Expert users might be interested in investigating these logs, e.g., to see which web pages could not be downloaded, and for what reason, or why the relevance score for a specific page could not be calculated. Accessing these logs directly via the system's frontend could offer a more user-friendly way to interact with the system's logs. The logs could be categorized in global crawler messages and messages related to a specific web page. This would allow to display the messages regarding specific pages directly on the frontend's results page, in the table that lists all downloaded pages for a crawling session.

7.5. Extended Configurability

While the proposed system provides a sufficient level of configurability and fulfills the requirements for its use case, there is always room for improvement and extension. During the test phase of this project, some possible features and quality of life improvements regarding configurability were identified:

- **Blacklisting –** By providing an interface that allows to mark domains and web sites as non-relevant, users could have more influence on the paths the crawler takes and help avoiding non-relevant content. Currently, this could be achieved by setting the disallowed pages of a domain in the database, but this approach is not really user friendly and would undermine the purpose of the disallowed pages column in the domains table, as it is intended to store paths on a domain that are defined in the domains robots.txt file.
- **Schema Updates –** The current state of the crawling system uses version 20.0 of the schema.org vocabulary. The system does not provide an interface that allows users to update used schema.org version. A simple step-by-step migration guide that lets users provide a schema.org definition document, identifies differences to the current version and lets users provide synonyms for new types and properties could help keeping the system up to date.
- **Hardcoded Values –** The internals of the system use an array of hardcoded values that were chosen during the time of development. When using the system in a production-like environment, a need for configuring these values might emerge. For example, the maximum number of worker threads that the crawler opens cannot be changed without editing the source code. By making this number configurable through the systems web frontend, experienced users could adapt the system to the environment it is deployed in, possibly enhancing performance. However, providing the possibility to change the internal configurations of the crawler also imposes the risk of breaking functionality. For this reason, implementing different user roles with different rights might be advisable.

8. Conclusion

This thesis presented the design, implementation, and experimental evaluation of a focused web crawling system. The system is built using the Java programming language, the Spring framework, and Jsoup and integrates various external APIs for relevance scoring and seeds acquisition. The frontend application, based on the Spring framework and Thymeleaf, provides an intuitive web interface for user interaction and a REST-API for programmatic interaction with the system. Communication between the frontend and the backend is facilitated through RabbitMQ, ensuring a decoupled, stable, and scalable architecture. Docker and Docker Compose have been employed for ease of deployment.

Addressing the challenges associated with traditional web crawling, such as scalability, content selection tradeoffs, and social obligations, a rule-based approach was implemented that effectively identifies and prioritizes relevant web pages by calculating relevance metrics which have been specifically designed to meet the requirements imposed by the use case. This rule-based approach has been augmented with the use of a Large Language Model, GPT-3.

The experimental evaluation highlighted the importance of refining the relevance metrics for clear distinction of relevant and non-relevant web pages. Initially, the HTML Score showed a high correlation with actual relevance, albeit not sufficiently discriminating. A sigmoid function was applied to improve this, successfully spreading the range of assigned scores while keeping – and in some cases, increasing – the correlation between HTML Scores and actual relevance.

The system serves as a robust proof-of-concept, demonstrating the feasibility and effectiveness of combining traditional rule-based methods with machine learning models like ChatGPT for focused web crawling. While the current implementation has room for improvement, it lays a solid foundation for future work in the field of web crawling and information retrieval, and extraction.

Bibliography

[1] R. V. Guha, D. Brickley, and S. Macbeth, “Schema.org: evolution of structured data on the web,” *Communications of the ACM*, vol. 59, no. 2, pp. 44–51, Jan. 2016, doi: 10.1145/2844544.

[2] M. Röhlig, “Österreich: Google legt sich versehentlich schon Hofer als Bundespräsident fest,” *Der Spiegel*, May 23, 2016. Accessed: Aug. 06, 2023. [Online]. Available: https://www.spiegel.de/netzwelt/web/oesterreich-google-legt-sich-versehentlich-schon-hofer-als-bundespraesident-fest-a-00000000-0003-0001-0000-000000582674

[3] “WorldWideWebSize.com | The size of the World Wide Web (The Internet).” Accessed: Jul. 21, 2023. [Online]. Available: https://www.worldwidewebsize.com/

[4] A. Akbar, S. Caton, and R. Bierig, “Personalised Filter Bias with Google and DuckDuckGo: An Exploratory Study,” in *Artificial Intelligence and Cognitive Science*, L. Longo and R. O’Reilly, Eds., in Communications in Computer and Information Science. Cham: Springer Nature Switzerland, 2023, pp. 502–513. doi: 10.1007/978-3-031-26438-2_39.

[5] B. J. Jansen, A. Spink, and T. Saracevic, “Real life, real users, and real needs: a study and analysis of user queries on the web,” *Information Processing & Management*, vol. 36, no. 2, pp. 207–227, Mar. 2000, doi: 10.1016/S0306-4573(99)00056-4.

[6] D. Lewandowski, “A three-year study on the freshness of web search engine databases,” *Journal of Information Science*, vol. 34, no. 6, pp. 817–831, Dec. 2008, doi: 10.1177/0165551508089396.

[7] J.-G. Lee, D. Bae, S. Kim, J. Kim, and M. Y. Yi, “An effective approach to enhancing a focused crawler using Google,” *The Journal of Supercomputing*, vol. 76, no. 10, pp. 8175–8192, Oct. 2020, doi: 10.1007/s11227-019-02787-9.

[8] T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret, “The World-Wide Web,” *Communications of the ACM*, vol. 37, no. 8, pp. 76–82, Aug. 1994, doi: 10.1145/179606.179671.

[9] T. Berners-Lee, J. Hendler, and O. Lassila, “The Semantic Web,” *Scientific American*, vol. 284, no. 5, pp. 34–43, 2001.

[10] “Einführung in die Funktionsweise von Markup für strukturierte Daten | Google Search Central | Dokumentation | Google for Developers.” Accessed: Sep. 03, 2023. [Online]. Available: https://developers.google.com/search/docs/appearance/structured-data/intro-structured-data?hl=de#structured-data-format

[11] “HTML Standard.” Accessed: Sep. 03, 2023. [Online]. Available: https://html.spec.whatwg.org/multipage/microdata.html#microdata

[12] “RDFa Core 1.1 - Third Edition.” Accessed: Sep. 03, 2023. [Online]. Available: https://www.w3.org/TR/rdfa-core/

[13] X. Zhou and R. Zafarani, “A Survey of Fake News: Fundamental Theories, Detection Methods, and Opportunities,” *ACM Computing Surveys*, vol. 53, no. 5, p. 109:1-109:40, Sep. 2020, doi: 10.1145/3395046.

[14] R. Oshikawa, J. Qian, and W. Y. Wang, “A Survey on Natural Language Processing for Fake News Detection,” in *Proceedings of the Twelfth Language Resources and Evaluation Conference*, N. Calzolari, F. Béchet, P. Blache, K. Choukri, C. Cieri, T. Declerck, S. Goggi, H. Isahara, B. Maegaard, J. Mariani, H. Mazo, A. Moreno, J. Odijk, and S. Piperidis, Eds., Marseille, France: European Language Resources Association, May 2020, pp. 6086–6093. [Online]. Available: https://aclanthology.org/2020.lrec-1.747

[15] M. Fernandez and H. Alani, “Online Misinformation: Challenges and Future Directions,” in *Companion Proceedings of the The Web Conference 2018*, in WWW ’18. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, Apr. 2018, pp. 595–602. doi: 10.1145/3184558.3188730.

[16] G. Pennycook and D. G. Rand, “Fighting misinformation on social media using crowdsourced judgments of news source quality,” *Proceedings of the National Academy of Sciences*, vol. 116, no. 7, pp. 2521–2526, Feb. 2019, doi: 10.1073/pnas.1806781116.

[17] S. Tschiatschek, A. Singla, M. Gomez Rodriguez, A. Merchant, and A. Krause, “Fake News Detection in Social Networks via Crowd Signals,” in *Companion Proceedings of the The Web Conference 2018*, in WWW ’18. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, Apr. 2018, pp. 517–524. doi: 10.1145/3184558.3188722.

[18] P. Kalchgruber, W. Klas, N. Jnoub, and E. Momeni, “FactCheck - Identify and Fix Conflicting Data on the Web,” in *Web Engineering*, T. Mikkonen, R. Klamma, and J. Hernández, Eds., in Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 312–320. doi: 10.1007/978-3-319-91662-0_25.

[19] C. Olston and M. Najork, “Web Crawling,” *Foundations and Trends in Information Retrieval*, vol. 4, no. 3, pp. 175–246, 2010, doi: 10.1561/1500000017.

[20] S. Chakrabarti, M. van den Berg, and B. Dom, “Focused crawling: a new approach to topic-specific Web resource discovery,” *Computer Networks*, vol. 31, no. 11, pp. 1623–1640, May 1999, doi: 10.1016/S1389-1286(99)00052-3.

[21] D. Arotaritei and S. Mitra, “Web mining: a survey in the fuzzy framework,” *Fuzzy Sets and Systems*, vol. 148, no. 1, pp. 5–19, Nov. 2004, doi: 10.1016/j.fss.2004.03.003.

[22] M. H. Syed and S.-T. Chung, “MenuNER: Domain-Adapted BERT Based NER Approach for a Domain with Limited Dataset and Its Application to Food Menu Domain,” *Applied Sciences*, vol. 11, no. 13, Art. no. 13, Jan. 2021, doi: 10.3390/app11136007.

[23] A. Brandsen, S. Verberne, K. Lambers, and M. Wansleeben, “Can BERT Dig It? Named Entity Recognition for Information Retrieval in the Archaeology Domain,” *Journal on Computing and Cultural Heritage*, vol. 15, no. 3, p. 51:1-51:18, Sep. 2022, doi: 10.1145/3497842.

[24] K. Vieira, L. Barbosa, A. S. da Silva, J. Freire, and E. Moura, “Finding seeds to bootstrap focused crawlers,” *World Wide Web*, vol. 19, no. 3, pp. 449–474, May 2016, doi: 10.1007/s11280-015-0331-7.

[25] A. C. Eberendu, “Unstructured Data: an overview of the data of Big Data,” *IJCTT*, vol. 38, no. 1, pp. 46–50, Aug. 2016, doi: 10.14445/22312803/IJCTT-V38P109.

[26] “Usage Statistics and Market Share of Content Languages for Websites, July 2023.” Accessed: Jul. 25, 2023. [Online]. Available: https://w3techs.com/technologies/overview/content_language

[27] S. Kim and B.-T. Zhang, “Genetic Mining of HTML Structures for Effective Web-Document Retrieval,” *Applied Intelligence*, vol. 18, no. 3, pp. 243–256, May 2003, doi: 10.1023/A:1023293820057.

[28] V. N. Gudivada, V. V. Raghavan, W. I. Grosky, and R. Kasanagottu, “Information retrieval on the World Wide Web,” *IEEE Internet Computing*, vol. 1, no. 5, pp. 58–68, Sep. 1997, doi: 10.1109/4236.623969.

[29] P. Patil Swati, B. Pawar, and S. Patil Ajay, “Search engine optimization: A study,” *Research Journal of Computer and Information Technology Sciences*, vol. 1, no. 1, pp. 10–13, 2013.

[30] B. Ding *et al.*, “Is GPT-3 a Good Data Annotator?,” in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 11173–11195. Accessed: Aug. 07, 2023. [Online]. Available: https://aclanthology.org/2023.acl-long.626

[31] “AC/DC,” *Wikipedia*. Sep. 10, 2023. Accessed: Sep. 10, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=AC/DC&oldid=1174743984

[32] “Los Angeles Lakers,” *Wikipedia*. Sep. 06, 2023. Accessed: Sep. 10, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Los_Angeles_Lakers&oldid=1174175427

[33] “A Clockwork Orange (film),” *Wikipedia*. Sep. 09, 2023. Accessed: Sep. 10, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=A_Clockwork_Orange_(film)&oldid=1174597602

[34] “Methode entities.search | Knowledge Graph Search API,” Google for Developers. Accessed: Sep. 08, 2023. [Online]. Available: https://developers.google.com/knowledge-graph/reference/rest/v1?hl=de

[35] S. Batsakis, E. G. M. Petrakis, and E. Milios, “Improving the performance of focused web crawlers,” *Data & Knowledge Engineering*, vol. 68, no. 10, pp. 1001–1013, Oct. 2009, doi: 10.1016/j.datak.2009.04.002.

[36] J. Cho, H. Garcia-Molina, and L. Page, “Efficient crawling through URL ordering,” *Computer Networks and ISDN Systems*, vol. 30, no. 1, pp. 161–172, Apr. 1998, doi: 10.1016/S0169-7552(98)00108-1.

[37] “Important: Spiders, Robots and Web Wanderers from Martijn Koster on 1994-02-25 (stdin).” Accessed: Jul. 27, 2023. [Online]. Available: https://web.archive.org/web/20131029200350/http://inkdroid.org/tmp/www-talk/4113.html

[38] M. Koster, G. Illyes, H. Zeller, and L. Sassman, *RFC 9309: Robots Exclusion Protocol*. USA: RFC Editor, 2022.

[39] Wikipedia, “Wikipedia robots.txt,” robots.txt. Accessed: Jul. 27, 2023. [Online]. Available: https://en.wikipedia.org/robots.txt

[40] Y.-B. Yu, S.-L. Huang, N. Tashi, H. Zhang, F. Lei, and L.-Y. Wu, “A Survey about Algorithms Utilized by Focused Web Crawler,” *JEST*, vol. 16, no. 2, pp. 129–138, Jul. 2018, doi: 10.11989/JEST.1674-862X.70116018.

[41] B. Novak, “A survey of focused web crawling algorithms,” *Proceedings of SIKDD*, vol. 5558, pp. 55–58, 2004.

[42] “Pricing.” Accessed: Sep. 07, 2023. [Online]. Available: https://openai.com/pricing

[43] “GPT-3.5 Turbo fine-tuning and API updates.” Accessed: Sep. 29, 2023. [Online]. Available: https://openai.com/blog/gpt-3-5-turbo-fine-tuning-and-api-updates

[44] “Researcher Access Program application.” Accessed: Sep. 29, 2023. [Online]. Available: https://openai.com/form/researcher-access-program

[45] H. Touvron *et al.*, “Llama 2: Open Foundation and Fine-Tuned Chat Models.” arXiv, Jul. 19, 2023. doi: 10.48550/arXiv.2307.09288.

[46] “GPT-4 API general availability and deprecation of older models in the Completions API.” Accessed: Sep. 29, 2023. [Online]. Available: https://openai.com/blog/gpt-4-api-general-availability

Acronyms

API Application Programming Interface

BERT Bidirectional Encoder Representations from Transformers

CRUD Create, Read, Update, Delete

CSS Cascading Style Sheet

DOM Document Object Model

HMM Hidden Markov Model

HTML Hypertext Markup Language

IE Information Extraction

IR Information Retrieval

JDK Java Development Kit

JPA Jakarta Persistence Application Programming Interface

JSON-LD JavaScript Object Notation for Linked Data

JSON JavaScript Object Notation

LLM Large Language Model

MIME Multipurpose Internet Mail Extensions

ML Machine Learning

NLP Natural Language Processing

ORM Object-Relational Mapping

RDFa RDF in Attributes

RDF Resource Description Framework

REST Representational State Transfer

RFC Request for Comments

SQL Structured Query Language

SSE Server-Sent Events

WWW World Wide Web

XML Extensible Markup Language

YAML YAML Ain’t Markup Language