Faculty of Science and Technology
Department of Computer Science

Schema crawled API documentation exploration

—

Nicolai Bakkeli

INF-3981 Master's Thesis in Computer Science - December 2017

Abstract

An increase in open APIs deployed for public use have been seen to grow rapidly in the last few years and are expected to do so even faster in the future. This thesis deliver a design that reduces the API documentation exploration process by recommending a range of suitable APIs for any user. This is done as a response to the increased complexity in API selection that follows as a consequence of the increased choice between APIs. This design suggestion consists of two main components; a tailor-made web crawler that collects API documentation and a handler that indexes the documentation and evaluates XML Schema for supposed API input and output searches. The services computational chain creates an overview on the API containing domains and a ranked list of APIs based on key-phrases applied by any user. Experiments of an implemented version of the service revealed that the indexation process creates a table that causes the service to run slower as the reach of the service grows. In other words, the indexed data stored on disk causes a scalability issue that does not get resolved in this thesis. Aside from performance problems have the service been shown to yield data that can be considered useful for developers in need of API recommendations.

Acknowledgements

First and foremost I would like to thank my thesis advisor Anders Andersen of the The University of Tromsø the Arctic University of Norway. Andersen's door has always been open for consultation whenever I ran in need of inspiration or guidance from someone with expertise. Finally, I would like to extend my gratitude to family, friends and fellow co-workers for your contribution in form of help and support throughout the construction of the thesis.

N.B.

(Nicolai Bakkeli)

Contents

Abstract **i**

Acknowledgements **iii**

Contents **vii**

List of Figures **ix**

List of Tables **1**

1 Introduction **2**
- 1.1 Problem statement . . . 3
- 1.2 Limitations . . . 4
- 1.3 Thesis Structure . . . 4

2 Background **5**
- 2.1 API . . . 5
- 2.2 REST . . . 5
- 2.3 REST API . . . 7
- 2.4 Web Crawling . . . 7
- 2.5 API Documentation Schema . . . 8
  - 2.5.1 XML Schema . . . 8
  - 2.5.2 JSON Schema . . . 8
- 2.6 Features of good API Documentations . . . 9
- 2.7 Related work . . . 9
  - 2.7.1 Automated Information Extraction from Web APIs Documentation . . . 9
  - 2.7.2 MAPO: Mining API usages from open source repositories . . 9

3 Approach **11**
- 3.1 Objectives . . . 11

3.2 Design Stage . . . 12
3.2.1 Data Acquisition . . . 12
3.2.2 Data Handling . . . 12
3.3 Implementation Stage . . . 13
3.3.1 Tools . . . 13
3.3.2 Creating Main Components . . . 15
3.4 Measurement, Analysis and Comparison Stage . . . 15
3.4.1 Result Extraction . . . 15
3.4.2 Testing Components and Experimental Setup . . . 16
3.4.3 Analysis . . . 16

4 Design **18**
4.1 Key Realizations . . . 18
4.2 User To Service Relation . . . 19
4.3 Data Acquisition . . . 20
4.3.1 API Documentation Discovery . . . 21
4.3.2 Domain Hierarchy Constructor . . . 21
4.3.3 Coordinator . . . 23
4.3.4 Communicator . . . 24
4.4 Data Handling . . . 24
4.4.1 Page Indexer . . . 24
4.4.2 Schema Evaluator . . . 26
4.4.3 Key Phrase Database . . . 26
4.4.4 Key Phrase Lookup Service . . . 27

5 Implementation **28**
5.1 Preparatory Work . . . 28
5.1.1 Testing Machine . . . 28
5.1.2 Test Domains . . . 29
5.1.3 Content Switch . . . 30
5.1.4 Error Reporting . . . 30
5.1.5 Service Configurations . . . 30
5.2 Crawler . . . 31
5.2.1 Hierarchy Constructor . . . 31
5.2.2 Coordinator . . . 33
5.2.3 Communicator . . . 33
5.3 Data Handler . . . 33
5.3.1 Page Indexing . . . 34
5.3.2 XML Evaluation . . . 35
5.3.3 Lookup Service . . . 35

6 Experiments **37**
6.1 Data Sources 37
6.2 Tests 38
6.3 Results 40
6.3.1 Experiment - A_1 40
6.3.2 Experiment - A_2 41
6.3.3 Experiment - H_1 41
6.3.4 Experiment - H_2 42
6.3.5 Experiment - T_1 43
6.3.6 Experiment - T_2 45
6.3.7 Experiment - T_3 48
6.3.8 Experiment - T_4 51
6.3.9 Experiment - T_5 55

7 Discussion **58**
7.1 Project Evaluation 58
7.1.1 Data Acquisition 59
7.1.2 Data Handling 60
7.2 User Service 62
7.3 Future Work 62

8 Conclusion **65**

Bibliography **66**

Appendices **70**

A Results: Data Acquisition **73**

B Results: Data Handling **81**

C Misc **83**

List of Figures

2.1 REST architecture: Show how client to server communication is outlined. 6

4.1 Data Acquisition: Overview diagram. 20
4.2 Data Handling: Overview diagram. 25

6.1 CPU drain during indexed crawl over 3 domains. 46
6.2 Equation 6.1: Speed calculation of json and usjon. 47
6.3 CPU drain during pure indexation of HTML files from 3 domains. . . 48
6.4 Memory consumption during indexed crawl over multiple domains. . 50
6.5 Memory consumption during pure HTML indexation. 51
6.6 Memory consumption during schema evaluation. 52
6.7 Runtime of indexed crawl based on Indexation Table size. 53
6.8 Runtime of pure crawl based on HTTP response size. 54
6.9 Runtime of pure crawl based on total runtime. 55
6.10 Runtime of pure indexation based on HTML file size. 56
6.11 Runtime of pure indexation based on Indexation Table size. 57

A.1 URI Hierarchy represenstaion of the api.met.no domain. 78
A.2 URI Hierarchy represenstaion of the developers.gettyimages.com domain. 79
A.3 URI Hierarchy represenstaion of the crawling www.datamuse.com domain. 80

B.1 Example results from the key phrase lookup. 82

C.1 Setup Installations 84
C.2 Outliers - Crawl Time, Indexation Table size. 85
C.3 Outliers - Crawl Time, response body size. 86
C.4 JSON Schema for Indexation Table validation. 87
C.5 JSON Schema for XML Schema Table validation. 88

List of Tables

4.1 Visited URIs translated to the URI Hierarchy. 22
4.2 Traceback of URI Hierarchy to find parent site of XML Schema. . . . 23

5.1 Testing Machine Specifications 28

6.1 Test domain statistics. 38
6.2 Short summary on test goals. 40
6.3 Summary of HTML validation. 41
6.4 Word validation on recommended APIs. 42
6.5 cProfile of indexed crawl with JSON Schema validation active. . . . 44
6.6 cProfile of indexed crawl withoutJSON Schema validation active. . . 44
6.7 cProfile of Indexing NO crawl. 45
6.8 cProfile of XML Schema evaluation NO crawl. 45
6.9 Statistical description of data from indexed crawl 47
6.10 Statistical description of data from pure indexation 49

1 Introduction

Development of web-services generally is a process that involves a lot of factors in order to be successful. A typical way of thinking when outlining a project is to separate key elements into smaller segments that in the end are puzzled together into one coherent unit. Abstraction is necessary for these scenarios and is effectively used to foster comprehension before details are made clear or even determined. Segments necessary for patching together a project are within their own right smaller projects that need to be planned, constructed and tested. Each of these steps is time-consuming and can be hard in many ways to achieve based on a range of factors. Luckily developers are aware that their need mirrors the need of others and that the service they are creating can be useful for others. Because of this, tools made by developers are more often made available for others to use. Services these tools supply can often be used directly as necessary segments in other projects. Utilizing already existing services instead of creating every piece of the picture can reduce development time and resource costs. This can in turn increase the development time focused on the new features instead.

The spirit of sharing APIs have within the last decade become common among developers [1]. Thus increasing the available options for borrowed functionality. Exponentially so, in fact, ProgrammableWeb.com a web site containing an API Directory for various APIs passed a content count of 17000 API entries in 2017 [2]. More specific modules that fit niche applications have grown in tandem with the steady growth in choice among open solutions. A downside to this is that selection of an API appropriate for a given project under construction has become a more time-consuming task, more so now than before. Some of the reason for this is a dilution among previously clear keyword and API descriptions. Resulting in unclarity due to the blend of seemingly similar APIs in the eyes of the outsiders. This in turn increases the time sorting out the viable APIs.

Reduction in time spent choosing the correct API can be done in many ways but this thesis will focus on one approach. Choosing an API often comes down to the knowledge of URI queries and what these represent. Reducing the time spent comparing request and response types among APIs are usually a time-consuming part of the process. Often, developers choose APIs that utilize queries that fit their design plans.

By following the assumption that a common way of choosing APIs are as follows: Designers search the internet for APIs that have potential. By first getting an overview on possible useful APIs the developers can through an iterative process of elimination weed out APIs that are less weldable into their own solutions. As many designs have requirements that are met by a series of possible open APIs, developers often choose APIs that present the least resistance in development. As a result, a common way of finding the right APIs comes down to manually interpreting query phrases for a variety of resources, to find the API with the least amount of development time spent on adjusting the preexisting code.

Choosing the right API can be a hard task and is often dependent on a correct interpretation of documentation for the APIs in question. A common factor for an APIs success lies in the documentation [3]. Time spent interpreting functionality of an API not used is ultimately wasted time.

This thesis goal is to present a suggestion on how to reduce the negative effect on the bottleneck of human reading speed and comprehension in regards to choice among open APIs specifically. Naturally, decreasing the amount of API documentation required to read will decrease the total amount of time spent reading the documentation. This thesis proposes a technique for reducing the sum of documentation pages by combining two key components:

1. A web crawler specific for REST API documentation pages that creates a hierarchy overview of a domain and keeps track of the schema connected to the given APIs found.
2. Utilizing indexation of the documentation and attribute extraction of schema to create a ranked list of API recommendations that correspond to a set of user applied key-phrases or words.

1.1 Problem statement

Design a set of automatic overview mechanics presenting resource location and relevance on key phrases related to open APIs as a mean to reduce development delay due *to open API documentation exploration.*

The service outlined in this paper should serve to help any end user in finding open APIs that fit their need. Specifically, the user should be able to apply a set of key-phrases or words and expect to get a list of viable API recommended in return. This thesis will outline a set of separate mechanics that collectively serve to create such a list. Entries in this list must include a score that reflects that APIs relevance relative to the key-phrases or words applied. Additionally, each entry must also include a URI that lets the user reach the API documentation that the entry is based on.

1.2 Limitations

This thesis will not reflect on the security aspect of the design or implementation of this service. System load on the domains visited will be ignored along with any "politeness" requirements set in place by the holders of the domains. For instance, some domains wish that their sites only receive requests at a given rate or that the pages do not get indexed by crawlers. Additionally, deprecated systems will be treated identically to operational APIs. The entire construction of the service will be done on one single machine so the service will be created as a centralized proof of concept. How the service can be repurposed for other goals and applications will not be discussed and the aesthetic presentation of the end results will have a low priority.

1.3 Thesis Structure

This thesis will contain a background chapter(**2**) that gives a brief summary on material relevant for the service that is to be constructed. The background will be followed by the approach chapter(**3**) that outline the workflow and methodology used to reach the goal of the thesis. Up next comes the design chapter(**4**) that given an explanation of one possible design to implemented in the implementation chapter(**5**). A experiments chapter(**6**) continues the thesis where experiments are conducted to test if the service act as predicted. The last two chapters(**7** & **8**) discusses whether or not the service acts in accordance with the problem statement and the final chapter summarizes the thesis.

2 Background

This chapter describes relevant technical background and relevant work. It will give an overview of concepts used in the thesis will be briefly explained. We will first define web crawling, and then describe the methods that have been used in this thesis.

2.1 API

Application programming interfaces (APIs) are routines, protocols and other tools designed for software development. These tools are often lists of clearly defined methods of communication between software components. Good APIs use abstractions of the underlying implementations to isolate exposure to the features that are deemed useful for future development. APIs are can be viewed as the name implies, an interface between an enterprise and applications that use its assets.

2.2 REST

In the late 80's to the yearly 90's Tim Berners-Lee a computer programmer working at the European Organization for Nuclear Research (CERN), in Geneva, Switzerland had some great ideas that ended up becoming the "WorldWideWeb" [4, 5]. This lead to the creation of the Uniform Resource Identifier (URI), the HyperText Transfer Protocol (HTTP), HyperText Markup Language (HTML), the first web server and browser. This all became so popular that the Internet infrastructure was predicted to collapse due to the popularity outgrowing the capacity. As the problem seemed to increase, Roy Fielding the co-founder of the Apache HTTP Server project [6] proposed some constraints that he grouped into six categories:

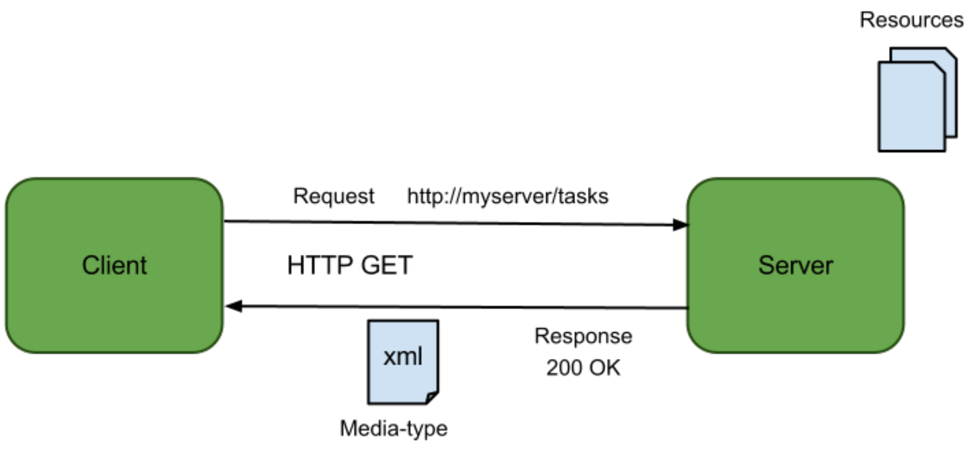


Figure 2.1: REST architecture: "In REST architecture there is always a client and a server where the communication is always initiated by the client. The client and server are decoupled by a uniform interface there by making both the client and server to develop independently. Every resource in the server is accessed by a unique address (URI). When the client access a resource the server returns a representation of the resource based upon the request header. The representations are usually called as media-types or MIME types." [7]

- **Client-server:** Separation between client and server, distinguishing them so they can be constructed, tested and deployed independently.
- **Uniform interface:** Separate web components need uniformity of their interfaces. These constraints are: Unique Web-based concepts should be considered resources accessible through some unique identifier, such as URI. Manipulation of resources through representation, turning a single resource into multiple representations while keeping a single identifier. Self-descriptive messages embedding intentions into messages through metadata. And finally Hypermedia as the engine of application state (HATEOAS), the claim that the impotence or use of a resource is to some degree determined by the inclusion of links to other resources.
- **Layered system:** Transparently deployed intermediaries that intercept client-server communication, usually to enforce security, response caching and load balancing.
- **Cache:** Caching previously received data to decrease required response loads

on upcoming requests. Increases overall availability and reliability and reduces the overall cost of the Web.

- **Stateless:** Web services are not required to remember the state of its client applications. To compensate, requests clients send must contain contextual information required for each interaction.

- **Code-on-demand:** Web services can send executable code to clients.

Using these constraints in some collaborated work, Fielding, Berners-Lee, and others managed to increase the scalability of the web, and in doing so standardizing the web. In the aftermath of the scalability crisis, Fielding named and described the Web's architectural style "Representational State Transfer" (REST) [8, 4, 9].

2.3 REST API

REST APIs are Web APIs that follow the architectural style of REST. This is usually achieved by having purpose-built web servers that handle function calls incoming through standard HTTP requests. Requests are created by clients that are in need of the servers responses and the communication is made available through URLs directed at the application server's resource. [4]

2.4 Web Crawling

Web crawlers also known as spiders [10] are automated browsing tools that work towards getting overview information on public content. Crawlers gather information on open databases, web pages, and resources by examining data downloaded from the source in question. A variety of crawlers have been created throughout the years [11] and most have some key-features in common. Content examined by crawlers are reviewed on a page by page basis. Page acquisition is automated and continuous to some degree. Upcoming fields to investigate are dynamically updated based on the findings on previously visited sites. Web crawlers typically start off with a set of URLs that it should visit, these are called seeds. Automatic download of these pages is followed by extraction of URLs form the websites HTML files [12]. New possible endpoints to examine are presented by the list of URLs found in the hypertext. Following new URLs with the same plan as previously creates a loop that feeds itself.

Typical motivations for using web crawlers are as follows:

- Content indexing for search engines.
- Automated testing and model checking of web applications.
- Automated security testing and vulnerability assessment [11].

2.5 API Documentation Schema

2.5.1 XML Schema

XML Schema is document files that describes the possible contents of XML files (Extensible Markup Language). These XML Schema are used in conjunction with XML documents to verify the contents of the structure of variables inside the XML documents. [13]. This is useful when XML files need to conform to a set of criteria set in place by the developer. By default are XML documents created without any associated schema but these schemas are sometimes used to explain the output of APIs.

2.5.2 JSON Schema

JSON(JavaScript Object Notation) is not in its original form built around any schema structure the way XML is. But there some modules that make up for this. One particular module named JSON Schema is commonly used to verify the structure of other JSON documents.[1]

[1]https://spacetelescope.github.io/understanding-json-schema/UnderstandingJSONSchema.pdf

2.6 Features of good API Documentations

Good API documentation should include a descriptive document that explains the API as a whole. It should consist of the logic behind the API and the intended use of the listed functions that the API contain. Additionally, the document should give an overview of the scope of the API and give the user a general idea on the usage in terms of patterns and expected behavior on both ends.

Good documentation should include a set of examples on how to interact with the API, in such cases is it often convenient for the user if output examples are presented as well.

Functions listed should also include data-types and other specs useful for the user.

2.7 Related work

2.7.1 Automated Information Extraction from Web APIs Documentation

An API documentation crawler made by Papa Alioune Ly, Carlos Pedrinaci and John Domingue aimed at RPC(Remote procedure call) and RESTful interfaces. The interesting segment of their crawler is the way the crawler processes HTML files. The authors utilize common traits in documentation design to their advantage, such as repeated chucks of layout that list HTML functions connected to the APIs. By going over the file to find patterns that repeat itself within the documentation can the crawler locate what the authors have termed block content. The crawler continues by looking through the content blocks to find HTML methods such as the GET, POST, PUT and DELETE keywords. [14]

2.7.2 MAPO: Mining API usages from open source repositories

A mining framework by Tao Xie and Jian Pei that builds upon existing search engines. The service they have created uses the search engine to extract source code from open source repositories through a query search. Source files extracted gets analyzed by a code analyzer that locates function calls throughout the code.

Then, the typical sequence of these function calls gets discovered. The result is an overview of the sequence of the functions typically used when using the API. And it is up to the user to determine which API fit their need through reading the function sequences. [15]

3 Approach

This chapter tackles methodology and outlines actions that will be taken to address the problem stated in the **Problem Statement 1.1**: *Design a set of automatic overview mechanics presenting resource location and relevance on key phrases related to open APIs as a mean to reduce development delay due to open API documentation exploration.*

3.1 Objectives

The main objective of this thesis is to reduce the open API documentation exploration by presenting a list of viable APIs that accept query key phrases similar to a target query search by any user. This will be achieved by using automated data acquisition on the API documentation pages. Acquired data will be examined and stored if deemed relevant. A separate algorithm that through a query key-phrase translation table will compare API documentation for query similarities that create a select line of viable API choices. Work done on this thesis is structured into three stages which will be explained in the upcoming sections. **Section 3.2** named the Design Stage will explain how the project was planned out and what would be assumed to be some required steps in order to fulfill the design goal from the **Problem Statement 1.1**. The next step **Section 3.3** will explain how the creation of the design was carried out in praxis and how the elements from the design ultimately came together to serve the purpose of the design. Lastly, **Section 3.4** will explain how measurements and analysis on these measurements are conducted. The purpose of the last stage is to create an overview of how well the end product is accomplished compared to the **Problem Statement 1.1**.

3.2 Design Stage

This subchapter describes the design process and the planning steps initially deemed necessary in order to start the implementation process that should solve the outlined problem. Subdividing this chapter further is an attempt at clarifying the future workflow where **Data Acquisition** will focus on how to get data that can will be analyzed and **Data Handling** focuses on how this acquired data can be analyzed and utilized.

3.2.1 Data Acquisition

The first stage of the design will concentrate on finding appropriate data that build the per-case groundwork for upcoming data analysis that could be used to construct a sound list of viable API choices. Finding appropriate technique(s) on how to discover open APIs with available API documentation will be the initial focus. The reaction to this problem will be explained in **Section 4.3.1** Following the fist initiative, treatment of sites containing API documentation should be outlined and a technical model on how to address attainment of the on-site information should be discussed. Automation of the key features should be addressed as valued component throughout the design stage. The design of the Data Acquisition is explained in detail in **Section 4.3**.

3.2.2 Data Handling

The second stage of the design should find a way to utilize collected information in order to produce API suggestions. This work builds upon the structure of the data acquisition so the design of the data handling should be devised once the former design is established. The Data handling aspect will be kept in mind although this phase is planned in succession to the data acquisition. This notion is key in fear of creating functionality that is difficult to build upon. Some key features that will be addressed in later in **Section 4.4**. These include among others how to filter files/data to focus on and how these files can be used to create an overview on suitable APIs for the end user.

3.3 Implementation Stage

Implementation is set up as a stage that follows after the design stage. It will consist of key work efforts to reach the design goals outlined in section **Section 3.2** and in detail in **Chapter 4**. This chapter, however, will explain which tools that were selected for implementation and on what round they were chosen. **Section 3.3.2** will include a rough summary of what functionalities the service will include. How these are constructed and how they work together will be explained in **Chapter 5**.

3.3.1 Tools

The right tools are crucial for the construction of the solution. Selection of tools was based on the usefulness of the tools in regard to how well the tool could serve the purpose of the task required and how time-consuming the tool would be to work with. The anticipated workload on coding is assumed to be high so any time efficient tool with the capability of others will be chosen above any competitor that yield the same result. Memory consumption and computational time is deemed as a lesser priority than the mentioned as the solution will act as a proof of concept.

- Python - The choice of programming language used for creation of the main components.[1] It is chosen because it is a high-level programming language that is widely used. Meaning that the code written for the project should be understandable for others to read and documentation for the language is solid. Additionally I the author are more comfortable with the language than with any other. Thus Python is the language that will create the least amount of resistance during development, which saves time.

- Graphviz - An open source graph visualization software will be used to visualize the URI hierarchy overview.[2] It is chosen because it can create vectorized graphics of the hierarchy. Additionally, there is a module of this software adapted for python.

- JSON Schema - Used for validation of any index databases created.[3] JSON is chosen because it's easily converted to python objects. The interesting part of it is to use it with python dictionaries that have an easy lookup scheme.

[1]https://www.python.org/about/
[2]http://www.graphviz.org/
[3]http://json-schema.org/

JSON Schema is chosen to make sure that JSON files used for indexation are written correctly or at least in an anticipated way.

- Tidy - Used to find errors in HTML code.[4] It is chosen because it makes sure that the HTML files collected are sound, which makes any extraction of data easier.

- Pypi Memory profiler - Python module for monitoring memory consumption.[5] Chosen because it is purpose-built for python and on the python wheel. It can also be run from shell which makes it easy to use and have a low learning curve. Additionally, its an outside process that does not get included in the measurements.

- Pypi Psutil - A cross-platform library for retrieving information on running processes and system utilization.[6] Also chosen because it is on the python wheel. It allows for measurement of CPU(Central processing unit) and disk reads and writes which might come in handy.

- NLTK - Natural Language Toolkit useful for counting words.[7] On the python wheel and useful for counting words in the XML and HTML files.

- matplotlib - Used for plotting of results.[8] On the python wheel and is easy to use. It will create all the plots for the results.

- NumPy - Fundamental package for scientific computing with Python.[9] Used to make the plots easier.

- Beautiful Soup - A Python library for pulling data out of HTML and XML files.[10] Chosen because it makes data extraction from HTML files much easier which reduces development time.

- USJON - Ultra fast JSON encoder and decoder for Python.[11]

- Lxml - Powerful and Pythonic XML processing library.[12]

[4]http://www.html-tidy.org/
[5]https://pypi.python.org/pypi/memory_profiler
[6]https://pypi.python.org/pypi/psutil
[7]http://www.nltk.org/
[8]https://pypi.python.org/pypi/matplotlib
[9]http://www.numpy.org/
[10]https://www.crummy.com/software/BeautifulSoup/bs4/doc/
[11]https://pypi.python.org/pypi/ujson
[12]https://pypi.python.org/pypi/lxml

3.3.2 Creating Main Components

Two main components will be constructed as a means to complete the entire service. The first set of algorithms will concentrate on collecting raw data that can be analyzed for API proposals further down the line. As an aim for this first set of algorithms, there will be created an automatic engine for traversal of API documentation sites. Utilizing a set of rules to distinguish what kind of data these pages consist of should yield appropriate responses for the data collected. Additionally, URIs visited should be nested together to create a relations-graph between the resources found.

The other set of algorithms should be responsible for constructing a set of API suggestions for any end user. This will include analysis of the collected data from the previous set of algorithms as a foundation. Sifting of user-defined key words trough the end result of the previous stage included a ranking algorithm will be the main inclusion of the last step. The last set of algorithms should also create a presentation for the end user of the APIs it deems useful for the end user.

3.4 Measurement, Analysis and Comparison Stage

Measurements will be conducted as the last stage of the work on the thesis. Data gathered throughout this segment will be thoroughly looked upon and analyzed.

3.4.1 Result Extraction

Intermediaries

The URI hierarchical graph will be created directly by the solution and will be examined manually. Python scripts will be created for the rest of the scripts that will create the test data. These scripts will include validation of HTML code and the indexation database as well as measurements on memory and disk usage. The scripts will also check the execution speed of the various algorithms in the solution.

End results

Results from the Data Handler will automatically return a set of possibly viable APIs for the end user. Because of this, end results will be evaluated by taking a manual look at the returned list. Scores on the APIs will be scrutinized by comparing inputted key phrases to the contents of the websites suggested by the Handler.

3.4.2 Testing Components and Experimental Setup

Experiments will be carried out by a physical machine running Ubuntu Linux 16.04 LTS. These experiments will be initiated by any deployer of the system and will not be initiated in large by itself during normal execution. Many of the tests are in themselves very resource costly and will, therefore, reduce the efficiency of the service they are trying to test. To work around this, execution of tests are done in closed environments that resemble real runtime.

After the creation of the main components, a set of tests checking the hierarchical representation should help to figure if the domains visited are thoroughly visited. Experiments concerning URI hierarchical visitation and Data Acquisition specially is labeled A_n. Following, experiments concerning Data Handling are labeled as H_n. Experiments on the whole solution is labeled T_n.

- Experiment A_1 - Manual comparison of hierarchical graph to web pages to confirm visitation route.
- Experiment A_2 - Check that the HTML files gathered are valid and without errors.
- Experiment H_1 - Validation of indexation database.
- Experiment H_2 - Manual execution of solution as end user and evaluate if API recommendations are reasonable.
- Experiment T_n - Various performance tests.

3.4.3 Analysis

The analysis stage can begin once data from the experiment stage are gathered to a reasonable amount. Analysis of the collected data will be done in-depth and should create a rough idea of how well the objectives are being met by the solution. This segment will include some speculation on the accuracy of the API recommendations by manual comparison. Naturally, this segment might be affected by biased views due to personal subjectivity. To reduce prejudice, the manual comparison will not judge the weight of the findings but merely prove the existence of the terms within the documentation pages.

Analysis of the systems resource use will also be done an this stage. This includes CPU and RAM use and how these are affected by a range of factors. The execution time of various algorithms will also be tested at this stage.

4 Design

This chapter displays how the tasks defined and outlined in the approach stage resulted and what is intended for implementation. The design to be explained act as a possible blueprint for automatic overview mechanics from the **Problem statement 1.1**. Later chapters will explain a way to implement this design and later on will these features be examined to see if they can be used to reduce development delay.

4.1 Key Realizations

It was clear after reviewing a series of API documentation pages that the means for programmatically deducing API Input and API Output should be handled in different ways to some extent. Input arrangements are often written in plain text and hold references to example queries for their core URL as tactics to demonstrate how to request data from the API. Output for the same APIs is often defined through plain text in style of the input descriptions but with some minor differences. At the same time, an attitude towards referencing schema as vehicles for giving their explanations further bearing are frequent and seems to be an accepted standard. These schema are resources separate from the documentation pages and this in turn pose as a clutch for a page by page evaluation design. Both schema and the API documentation page it was referenced from should be considered together in order to properly establish what the output from the API presents.

Different types of schema are used to explain API output where XML Schema is the most common type of schema seen so far. This is not to say that XML is the most common data format for return values, but are among the types that utilizes schema for control of data fields and types. Another form of schema is the JSON Schema that is not used that much as JSON was built without the schema utility in mind. For the purpose of this project, XML Schema will be the only type of schema evaluated in order to create a coherent product within the timespan. The inclusion of JSON schema can be done as future work.

Input for the API cannot be drawn from the XML Schema so the plain text and query examples will act as the entire foundation for the input evaluation. In order to get this data, the solution will extract this information from the HTML code received from the documentation pages.

Additionally the range in which API documentation websites are constructed varies a lot. Different API developers have different ideas as to how their documentation should be presented to any user. Some even let a large portion of their API be documented through forums, Q/A sections or even blog posts [16]. This causes a wide variety of interlinking between websites to create certain API documentation. In this work, each website will stand on their own ground as a representation of a single API. Schema will however be connected to at least a single website that can represent the schema.

4.2 User To Service Relation

The user service this design tries to achieve is to look up the API documentation pages for data fields or descriptions that are similar to that of a list of key phrases submitted by the user. In doing so the design is to compare the key phrases provided by the user with the data found in the documentation. By analyzing similarities between the two, the design aims to measure the contrast in the fields to see if they correlate with the user's interests. In the end, the service delivers the most supposedly attractive APIs and their locations(URLs) to the user with a score on how they match to the keywords applied as the input to the service.

Further more, the convenience of the utility is only realized if the service is faster than the user is in doing the same job manually. So the ideal situation is to do the wast majority of the work ahead of the involvement of the end user. Thus the forthcoming explanation of the design will be discussed in the Data Acquisition and Data Handling section. In the Data Acquisition section will the prerequisite workload and design be outlined. Any user of this design section will be referred to as a "deployer" as the section is outside the realm of the end user. The following section is termed Data Handling and will outline the design that features intractable by the end user.

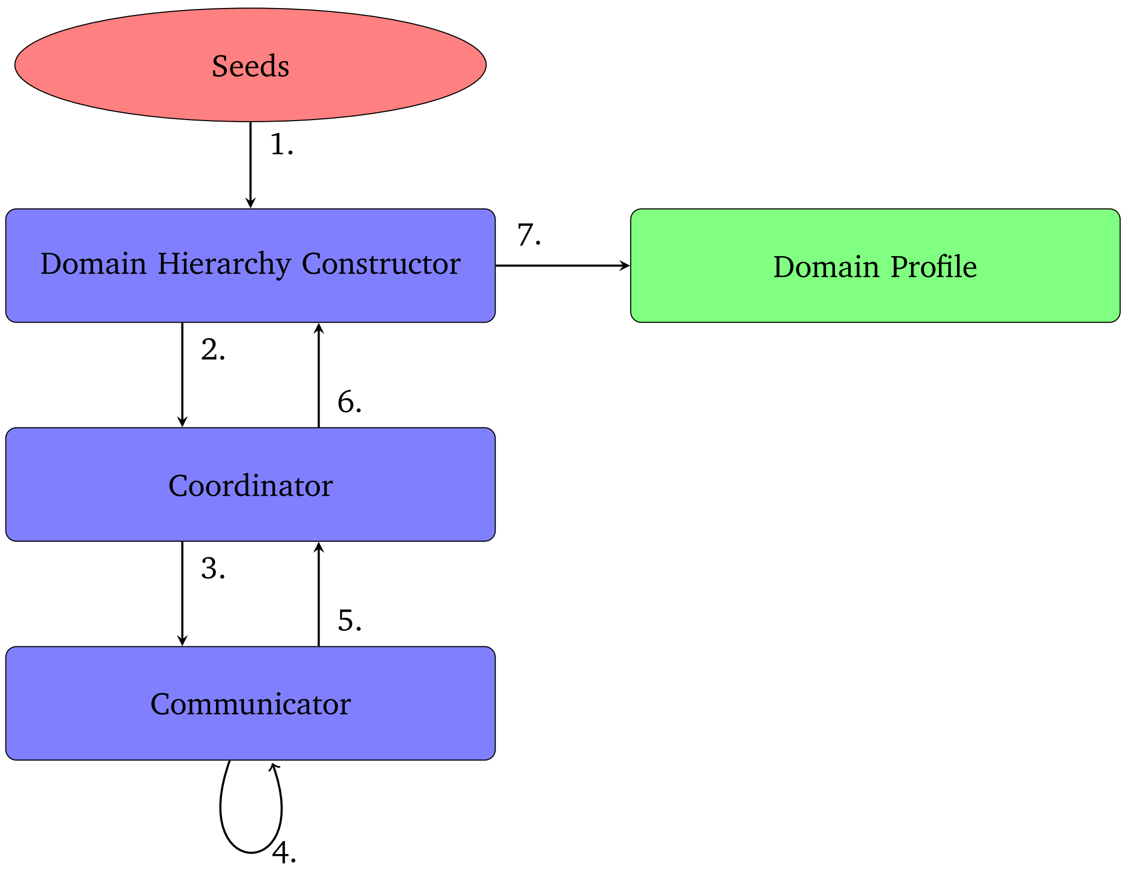


Figure 4.1: Data Acquisition: Overview diagram.[1]

4.3 Data Acquisition

This sub-chapter explain the design of the first main component mentioned in **Chapter 3.3.2**. Its clear after looking at different ways to acquire API documentation that if the data attainment were to be automated, then the process must be acknowledged as web crawling. As web crawlers can be created in a myriad of ways and have each their own purposes, the crawler for this project must be somewhat restricted in some aspects. The crawler must as well be specified to accomplish data acquisition regarded as fit for examination in the next step (Data

[1](**Figure 4.1** & **Figure 4.2**). Ellipses with the red color are data that originate from user input, blocks with the blue color display computational components of the model and blocks with the green color represent output or result data in some form.

handling). The Data Acquisition is further compartmentalized into three sections, these are the Domain Hierarchy Constructor, the Coordinator and the Communicator. These segments will coexist as separate units while as a whole be referred to as the crawler. An overview of the crawler can be viewed in **Figure 4.1**. There are seven steps in the overview diagram as seen in the Figure that represent the Data Acquisition. These numbers indicates the information flow between the segments of the crawler that will be described shortly.

4.3.1 API Documentation Discovery

One of the restrictions of the crawler for this design is that regular web crawlers are often able to visit a wide variety of domains in order to index the pages they come across. The crawler addressed in this chapter will focus on handling documentation pages. A distinction between API documentation pages and any other page on the World Wide Web is necessary in order to let the crawler lose to its full extent. This feature required to let the crawler sweep freely on the web will not be included as a design element. To work around this, a seed list containing URLs(seen as point 1. from the **Figure 4.1**) for a set of domains with web pages consistent of API documentation pages will be created. This seed list must be manually modified and updated by the deployer in order for the reach of the service to expand. The restriction of the crawler is that it keeps traversal local to the domains cataloged in the seed list. Any URL discovered that bridges the domain list will be discarded as an option of further traversal.

4.3.2 Domain Hierarchy Constructor

The Domain Hierarchy Constructor is in a way the head component of the crawler. The constructor can be seen in the **Figure 4.1** as the segment that receives URI seeds from the seed list and conducts all other communication with the Coordinator that will be explained soon. Additionally, the constructor is set to create the Domain Profile that is a collection of every stored HTML, XML file and the URI hierarchy that will get explained as well. The Domain Hierarchy Constructor is also tasked with keeping track on what pages to visit and the creation of a visual representation of how these are connected. The root cause of action is through a deployers demand. Upon initiation, it will gather a URI seed that functions as the entrance point for the crawl. By parsing the URI for its domain section the Domain Hierarchy Constructor locks its territorial operation to the website's root. Basing this initial URI as the first entry point to the crawlers visitation list creates the foundation for further operation. This list of unvisited websites is curated throughout the crawl. On the flip side, another list of visited URIs snatches any URI visited by the crawler. Examination of the visited list averts the crawler from visiting a page more than once. This is done to prevent unnecessary vitiations of know websites and creation of duplicates.

URI Hierarchy

The Domain Hierarchy Constructor creates a nested path hierarchy representation created from a subdivision of the visited URI paths. This entails a directory translation of the URIs path relations that device an unrealistic yet useful impression on how ownership between resources are set up. Schema as mentioned earlier are separate resources from the documentation pages. By backtracking through the URI Hierarchy to find the closest HTML location can the schema and HTML page be connected. **Table 4.1** show a short example on how a few searches get structure into a URI Hierarchy, where 3 of 4 websites yielded response data. While **Table 4.2** show how the XML Schema from URL "www.domain.com/path1/path2/path3/" gets paired with the HTML page from "www.domain.com/path1/". Pairing the schema to the documentation page is useful for later when the two are getting evaluated together.

URI Path Separation		
URI	subdivisions	response type
www.domain.com/	None	.html
www.domain.com/path1/	path1	.html
www.domain.com/path1/path2/	path1, path2	None
www.domain.com/path1/path2/path3/	path1, path2, path3	.xsd

URI Hierarchy
Domain: [path1: [path2: [path3: .xsd]], .html], .html

Table 4.1: Visited URIs translated to the URI Hierarchy.

Constructor information Exchange

New data and URIs are required for the crawler to continue and for the realization of the URI Hierarchy. Required information is collected from the Coordinator segment see chapter 4.3.3 (Coordinator). Attained information includes a list of new

XSD Parent Search		
Traceback	Focal Point	HTML
Domain: [path1: [path2: [path3: .xsd]], .html], .xml	path3	False
Domain: [path1: [path2: path3], .html], .html	path2	False
Domain: [path1: path2, .html], .xml	path1	True

XSD Location: www.domain.com/path1/path2/path3/
XSD Parent HTML Location: www.domain.com/path1/

Table 4.2: Traceback of URI Hierarchy to find parent site of XML Schema.

URIs that get added to the list of unvisited URIs under the criteria that they are not previously visited. Data flagged for storage are also written to disk at this point. All of this can be seen in the **Figure 4.1** where the second stage gives the Coordinator a URI, while the Coordinator answer with the mentioned data as stage six.

4.3.3 Coordinator

The Coordinator act as the choice taker for the crawler and is positioned between the Domain Hierarchy Constructor and the Communicator. A URI is transmitted to the Coordinator from the Domain Hierarchy Constructor Whenever the crawler attempts to visit a new website. This URI is forwarded to the Communicator(point 3. in the **Figure 4.1**) that return a response code and response data form the actual website communication, more on this later. Data acquired this way gets sifted through a response switch to determine how to react to the new data. What the Coordinator will not include that Papa Alioune Ly, Carlos Pedrinaci and John Domingue included are to locate common trait in the response to increase the accuracy of the extracted data, see **Chapter 2.7.1**. Neither will it utilize mechanics for extracting HTTP methods from the return bodies.

Content Switch

Content-Type header fields from RFC 822 [17] are accessible information paired with the return data. Control flow is forked based on these Content-Type header fields. Among these, the ones of interest are the application and text types. Detecting an application type start an extraction of subtype. A combination of these can determine whether or not the data is of the file type XML or XSD. In other words the discovery of XML Schema is done in this step. These files are flagged for storage or can be examined on the spot. DTD(Document Type Definition) subtypes are flagged for conversion to XML Schema. Additionally a Content-Type header field of text usually refers to plain text or HTML code and in some cases XML files. In cases of XML, are the schema location extracted. The reaction to plain text and HTML code is to extract any URI. Fragments in these URIs are trimmed away and the resulting URI is added to a list of new URIs that will be added to the Domain Hierarchy Constructors visit list.

4.3.4 Communicator

The communicator is the crawlers connection to the World Wide Web (point 4. from **Figure 4.1**). It receives the URI is should request data from through the Coordinator. By using HTTP requests the Communicator gets on website information that the other segments work upon. It parses response data to make it accessible for the other segments.

4.4 Data Handling

This sub-chapter explain the design of the second main component mentioned in **Chapter 3.3.2**. Data Handling refers to treatment of the collected data accumulated throughout the Data Acquisition stage. In reality this stage can be incorporated into the previous stage but this section makes more sense explained as a separate branch. The entirety of a catalog of data gathered from a domain will be referred to as a Domain Profile. There will in praxis be multiple Domain Profiles and can be seen in **Figure 4.2**. Do note that the Domain Profile form **Figure 4.2** is the out put from the crawler as seen in **Figure 4.1** as stage seven.

4.4.1 Page Indexer

The Page Indexer is one of the two first steps in the Data Handler chain see **Figure 4.2** point two. The Page Indexers job is to create a lookup table of words that yields the words source and occurrence rate. To compile such a table the Indexer starts out by loading HTML pages from a Domain Profile or gets it directly form

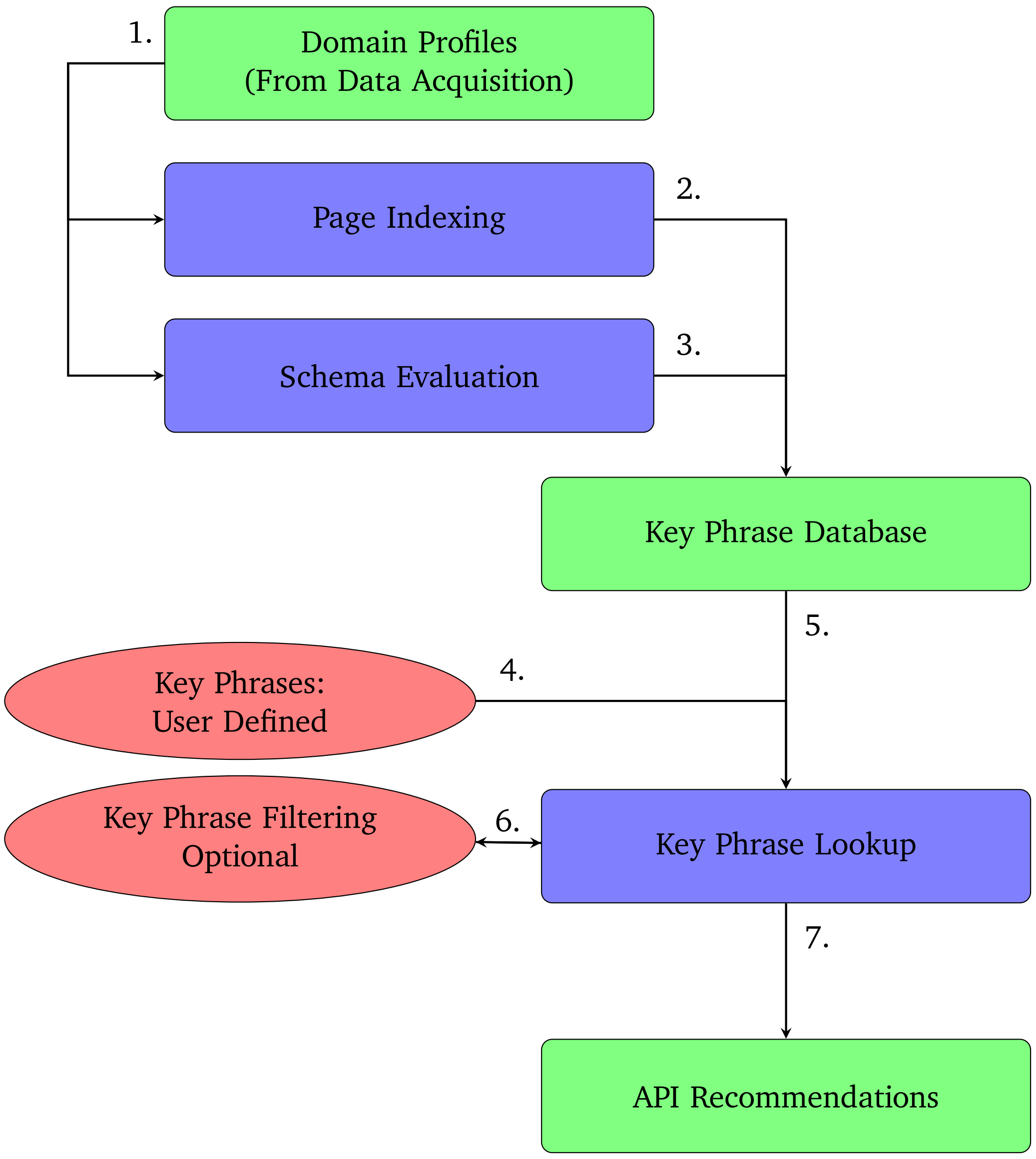


Figure 4.2: Data Handling: Overview diagram.[1]

the crawler at the crawl stage. Initially, the Page Indexer starts out by doing some preparatory work on its current page. Strings of text containing the key separators such as the following characters: : ? # [ ] @ ! $ & ' ( ) * + , ; are truncated and added as additions to the list of words to index. The original words are still kept in their entirety as well. This makes the lookup of words more precise at a later stage. The indexing can begin once the preparation is complete. The actual indexation iterates through the file and counts each word. Words found and their count are tacked onto the lookup table with the source of the document included so the source can be referenced upon lookup. Results from each file examined this way are stored in the same lookup table. A focus here is to make the word in the table easily accessible and through that word entry can the sources including the word be found.

4.4.2 Schema Evaluator

The Schema Evaluator is the other initial step of the Data Handler and can be seen as point three in **Figure 4.2**. Evaluation of schema is done in a somewhat similar way to the Page Indexing but there are some key differences. Attributes and elements from the schema define what the corresponding files are supposed to contain. This way schema is constructed to keep its length to a restricted and with a minimum of redundant repetition. As a consequence, there is no real benefit in counting the recurrence of fields found in the schema. On the other hand, some schema includes some text fields that are not intended to police what can be stored in the responding file. These can be counted for occurrence but the Schema Evaluator does not take this into account as the result table is structured in another way. Resulting terms are appended to a table that given a word yield every source containing the word.

4.4.3 Key Phrase Database

A combined effort from the Page Indexer and the Schema Evaluator crates the Key Phrase Database. These combined result records are stored to disk and form the backbone for the future lookup service. Creation of this database is done outside the end users reach and must therefore be handled by the deployer. This data collection that consist of the Indexation Table and the Schema Table can be seen in green in **Figure 4.2**.

4.4.4 Key Phrase Lookup Service

The Key Phrase Lookup Service seek API documentation pages that conform to a set of predetermined key phrases or words. These phrases or words must be applied by an end user that want to request a list of APIs that fit their need. Choice of key phrases is essential for the accuracy of this service. In order to extend the range of the search, the Key Phrase Lookup Service goes through each word and looks up synonyms and similarly spelled words. These are presented to the user that can filter through these words to make sure that any unwanted words get discarded. Finding synonyms can help the user in finding words that might be included in the database. This should make the chance of discovering any words from the database more likely. The drawback is that is dilutes the precision. Presenting similarly spelled words might help the end user with the spelling of the keywords. This might be useful if the end user has misspelled any words. Two separate lists of words are created this way, one for input and one for output.

Once this is done, the lookup will use the list of input keywords in conjunction with the lookup table created by the Page Indexer to generate the input scores of the API Documentation pages. This is done by iterating through the input key phrase list and finding every website that contains that word. Every website that uses the word gets included in a summary of websites where their word gets stored with its occurrence count. When it is done with every word in the key phrase list, the Key Phrase Lookup Service will continue by compounding each websites total score. The calculation of the output score is calculated in its own cycle. With a similar process to the calculation of the input score, the output calculation does not include the number of repetitions of each given word. To compensate, each word found have a higher intact on the score.

Additionally, for both of these score calculations words similar to the keyword from the Key Phrase Database gets included. These stray words get presented to the user that gets to chose to include them or not. Words included this way gets a similarity score attached to them. This similarity score is a point system that indicates how close they are to the original word. The inclusion of these words influences the total score for a given website by a smaller impact based on this similarity score.

Combining both input and output scores calculated creates the resulting basis. In order to present the finding to the end user, the Key Phrase Lookup Service sort the websites on the total score and presents it to the user. An example of such a presentation can be viewed in **Figure B.1**

5 Implementation

This chapter explains the implementation process of an example program that resembles the design outlined in the previous chapter, **Chapter 4**. However, this chapter starts out by explaining some preparatory work that is aimed at making the implementation process easier. Following subchapters will explain how smaller less abstract elements are put together to create the larger segments that as a whole manifest as a possible solution to the **Problem statement 1.1**.

5.1 Preparatory Work

5.1.1 Testing Machine

Construction of every component in solution and all testing was done on the same machine. Every test was intentionally done on the same machine in order to keep the results within boundaries that can be compared without any hypothetical translation between results and hardware alteration. Specs on the machine are listed in **Table 5.1**. See **Appendix C.1** for information on software.

packard bell EasyNote TK85	Hardware Specifications.
CPU	4 cores @ 2.53GHz
RAM	5,5 GiB
OS: Ubuntu Linux 16.04 LTS	

Table 5.1: Testing Machine Specifications

5.1.2 Test Domains

During the implementation of the service, some domains for the crawler to crawl through were needed. Three domains containing API documentation were discovered through a normal search on the web. These websites were chosen on various grounds and the choice of each of them are explained as follows:

1. http://api.met.no (About to be deprecated)
   Documentation sites for data collected by the Norwegian Meteorological Institute. Chosen because its domain contains a series of API documentation websites that use XML Schema for output explanation. The domain is also structured to be relatively shallow, thus keeping a branch of sub-paths low. Multiple APIs are described within this domain that is described in their own documentation sites.

2. http://developers.gettyimages.com
   Documentation sites for a stock image REST API by Getty images. Chosen because the domain really only contains a single API that is described through a series of different "child" sites. The structure of the domain is also deeper than that of api.met.no. Response bodies from the API are also mostly JSON files. Examples of these are however written in plain text in the HTML code, making the examples indexable.

3. https://www.datamuse.com
   Lastly, datamuse is an API for word-finding query engine documented through the domain. Its the smallest of the three and it keep its depth and width short.

An API documentation domain that got left out was the National Road Database from Norwegian Public Roads Administration.[1] The API contains a verity of documentation for utilities that among other things give information on objects on the road, maps and layouts of the roads in Norway. The domain was not included because it uses too many instances of fragments in their URIs which are stripped away in the implemented crawler. This coursed the domain traversal to terminate prematurely.

[1]https://www.vegvesen.no/nvdb/apidokumentasjon/

5.1.3 Content Switch

The volume of code that was to be created was anticipated to be large. And the number of separate files could also be many, so the first stage of content creation was directed at creating a content switch that let the testing and execution of every utility to be run from one single source. This switch generates a list of options for the user that can be activated through user input. The code magic in this utility is that the script gets hold of the names of every function it bestows. By sorting out every function with function name starting with the sub-sting "choice_" it finds every function that should be included in the presentation. These are represented by sorting the functions by function name and displaying the doc string of the functions as explorations to the switch alternatives. The aim of the switch was to have a utility that made content creation easier and to serve as a minimalistic UI for any end user.

5.1.4 Error Reporting

A wide variety of scenarios were expected from crawling the web. This creates the diverse range of potential errors that can occur during the crawl. Many of these failures are completely unexpected prior to running the crawler. This, in turn, makes the crawler prone to premature termination. A typical response would be to let the code run until it fails and fix the problem. Since the crawler would take some time to get back to the same problem was the creation of a debugging tool necessary. A script that when encountering an exception would parse the error and store the message with the in script location of failure were created. This script creates a file that only keeps unique entries that is easy to read. The main priority of the script is to let the crawler continue without interruption so errors could be handled in bulk.

5.1.5 Service Configurations

A configuration file was created to keep global variables to a minimum. This file serves as a common configuration file for the entire system. Alterations of the file are added as an option to the content switch so the user can change operational presets for the execution. Additionally, the URI seed list containing domains the crawler must crawl through is present in this file.

5.2 Crawler

The crawler is initiated through the content switch. Upon initiation will the crawler gather the seed list from the configuration file and start feeding them to the Hierarchy Constructor. This continues as long as there are uncrawled seeds in the list. Hierarchy representations returned from the Hierarchy Constructor are forwarded to a script that that translates the representation to DOT (graph description language) format. This is done in a recursive fashion where nodes created for the DOT scheme are URIs from the actual hierarchy. These nodes are connected between each other to make a tree graph that visualizes the domain hierarchy. Each node has hyperlink inclusion to make the website lookup easier. In the end, will the script translate the DOT file into a pdf that is ready for the user to look upon. The inclusion of this feature is intended to make the traversal of the crawled domain easier. Some results are expected to score some sibling pages to a similar weight. In these cases can valued information be extracted by looking their relation through this tree. Neighboring and parenting websites can also be found this way which can be of future use. An example of such a graph visualization of a hierarchy can be seen in **Appendix A.2** (the example does not include hyperlinks).

5.2.1 Hierarchy Constructor

The Hierarchy Constructor is present in a script that handles a single domain at a time. This script creates a list of URIs it is supposed to visit. A counterpart to this list is a list of visited URIs that consume the current visited URI from the visited list upon visitation. Initially is the URI seed the only URI in the visitation list and the crawler will work its way from that point. The Hierarchy Constructor will run its course until every URI in the visitation list is visited. Each entry in the visitation list is unaltered and might contain unwanted queries or fragments. A shaving routine is implemented to avoid using any unwanted endpoints among the URIs. This routine removes any fragments from the URI and will simply skip any URI that utilizes any form of query. The routine is initiated upon the URI swap that updates the current URI to a new one from the unvisited list.

Interaction with the Coordinator gives the Hierarchy Constructor the data needed to continue its work. How this data is gathered is explained in the next **Chapter 5.2.2**. However, the data gathered is a list of URIs present in the visited website and the actual data body. If any such data are returned are two sets of routines activated. The first routine is to update the visitation list. These links go through a netloc switch that determines if the URI belongs to the current domain. Separate handlers are devised to react to any link that follows relative paths, are part of the domain or from an outside source. Any URI pointing to an outside source are simply discarded, relative paths are united with the source URI and new URIs within the domain gets added to the visitation list.

Discovery of a new URI that is supposed to be visited starts a routine that creates or updates the URI Hierarchy. This is done by having a master hierarchy for the entire domain that is constructed as a nested python dictionary. Keys in these dictionaries are sub-paths from the URI discovered and the values are dictionaries with the same scheme. In order to update or create this hierarchy scheme, a routine will section the URI into several sub-paths that it recursively iterates. By diving into the nested structure sub-path by sub-path can the routine confirm if the sub-path is represented in the hierarchy. Any unrepresented sub-path will be added to the dictionary at the current depth.

This hierarchy dictionary is further used to create an on-disk directory tree constructed to be outlined the same way as the hierarchy dictionary. This on disk directory tree will be referred to as the physical dictionary.

Reaction to the data body is a set of routines that are queued after the URI list is examined. How this data is managed can be determined by a set of flags set in the configuration file. These flags are adjustable through the content switch. Some of these reactions such as indexation of HTML files can take place at this point but are described in the next **Chapter 5.3**. Routines react based upon the flags by storing the data contents of the data body into files located in the physical dictionary. This is done to index the files or to evaluate schema later. Files stored this way have information documents created as supplementary information sources on a file by file basis. These information files contain information on where the file came from (URI), its type and subtype if any. Some of the files stored this way have a source that is not represented by the URI it came from. This can for instance be XML files where the documentation page is a separate website. The parenting website is attempted to be found by iterating backward thorough the physical dictionary until an HTML file is located. By assuming that this file is the source of the XML file are the two connected for further unified examination.

Each URI source of any HTML file discovered will be stored in an HTML flag-table. This flag-table show whether or not an HTML source have been indexed. Lastly, a conversion tool translates XML file with the DTD subtype into XML Schema [18].

5.2.2 Coordinator

The real use of the Coordinator stems from the response it gets from the Communicator. This is done by forwarding a URI from the Hierarchy Constructor to the Communicator that returns some response data. Response data include a content type of the response body, a response body and status code from the communication. For each type of content type is there a content handler. Each of these content handlers is included in a type switch that creates a different reaction for each type of data body. These content handlers have each their own set of rules that determine what the crawler will do with the data body. The handlers that are of interest are the application and text handlers as they react to HTML and XML code. One of the jobs of the handlers are to determine the file extensions suitable for the data bodies. This is done by using a translation table that translates the file type into a file extension. Whenever a handler comes across an HTML file this way will a routine that extracts links from the file begin. This is done by using regular expression to find any links in the data body. Additionally, every tag in the HTML code is collected. Links located in these tags are extracted. The last step is to include relative paths if there are any. Links found this way are the URIs that Hierarchy Constructor examines later.

5.2.3 Communicator

Upon its initiation will the Communicator receive a URI from the Coordinator. Communication with the URI endpoint is done through pythons urllib module that yield the Communicator an http response. Any redirections that might occur are handled by this module if any. The response is parsed by the Communicator to get the response code and data body. Additionally, a handler extracts parameters, type, subtype and charset for the data body.

5.3 Data Handler

Initiation of the Data Handler is done in a similar fashion to the initiation of the crawler. However, the data handling is separated into three choices: schema evaluation, page indexing and key phrase search. Running the schema evaluation and the page indexing will collectively create the key phrase database as discussed in the design **Chapter 4.4.3**.

5.3.1 Page Indexing

Page indexing can either be initiated through the crawler when a data body form an HTML file is discovered or through the content switch. In the case of the crawler, the raw data is sent to the Page Indexer for examination.

While the content switch will open un-indexed HTML files located in the physical dictionary. Data contents opened this way will be fed to the Page Indexer iteratively so that every un-indexed HTML file gets indexed. The content switch gets hold of any un-indexed file by looking them up through the HTML flag-table described in **Chapter 5.2.1**

The first action of the Page Indexer is to look up the HTML in the flag-table to see if the HTML file is previously indexed and if so, skip it. If it is established that the HTML code is to be indexed will the Page Indexer load an indexation table into memory. This table will be referred to simply as the table. This table is created by the Page Indexer if there is no present table. The way the table is structured is to have key-value pairs where the keys words from the indexed files. Values are a new structure where a website is represented by the count of the word and URL of the website. This table if it exists will contain key-value pairs for all the words that occur in all the indexed HTML files.

Words from the HTML file is read from the file using Beautiful Soup. These words or phrases include in some cases example URIs. In these cases are some of the words, queries or other values connected. A separating scheme takes these URIs and fine grains them by splitting them based on a list of separators. The originals are however not lost. This concludes the preparatory work of the indexer.

The nltk module is used to get the vocabulary frequency of the words or phrases collected from the splitting scheme and HTML text. Results from this practice is then merged with the dictionary structure loaded from the table. The outcome is then converted to JSON an object that is written to disk as the newly updated indexation table. Upon completion will the HTML file be flagged as indexed in the HTML flag-table.

During implementation was a custom JSON Schema utilizes to enforce correctness in the indexation table. Use of this feature has since been dropped.

5.3.2 XML Evaluation

The XML Evaluation starts out by finding every information document created by the Hierarchy Constructor see **Chapter 5.2.1**. This is done through recursive traversal of the physical dictionary while picking out every file with a predetermined filename ending. These information documents include file type and subtype for the XML and XML Schema documents that the evaluator intend to evaluate. This helps the Evaluator skip the DTD files while picking up the translated counterpart of the file. Additionally the parent source URI of the file is extracted, this lets the XMl Schema score to be grouped with the parent HTML score.

An reader class is initiated that iterates through the XML files once the preparatory work is finished. This class is constructed to do the entire evaluation of every file imported. Evaluation done by the evaluator is primarily aimed at XML Schema but some responses stored by the crawler have a pure ".xml" file extension. Files ending with this file extension are also handled since some of these XML files include a source to their schema. By visiting their schema source some yield an XML Schema that can be evaluated, and will be instead of the XML file. This feature should have been included in the Coordinator as it forces the Evaluator to do web communication that is not suited here.

Evaluation of an XML Schema is done by recursively iterating through the file and extracting any raw text bodies form the file if any. These are added together as a total master string. For every attribute throughout the search will be added to its own list. These elements often contain combined words that the evaluator splits so each sub word can be evaluated separately. A shaving routine will remove any unwanted substrings from the text bodies once the word extractions are complete.

Resulting words and phrases are stored in a structure that keeps track on the sources of each word. This structure is in the end converted to a JSON object that gets written to disk.

5.3.3 Lookup Service

The Lookup Service can start once the API Documentation Acquisition is complete and either or both of XML Schema evaluation and HTML Page Indexing is done. Target words by the end user are extracted from the configuration file. These words are sectioned into words or phrases used for input and output. A common list of phrases is also devised that will be added to both the input and output register. A routine that finds synonyms and similarly spelled words goes over the lists in order to expand the range of the words. The API used for this service is one of the API sites used for testing and are called datamuse.[2]

Starting out with the indexed HTML data, the lookup service will start out by ranking the pages previously indexed. This is done by initially reading the indexation table into memory. A routine that iterates through every key phrase or word specified will follow after the table is in memory. During this routine will every word close to the current search word be extracted from the indexation table. These words are not always exactly the words used in the input list but are relatively similar. A score based on the similarity between the words are calculated as a ratio calculated by a sequence matcher. This calculation is then presented to the end user that gets to chose to include the word. Automation of the last step is togglable by the user so the end user does not have to answer as many questions. Turing it off will however reduce the accuracy of the service.

Every word found this way are added to a dictionary that keeps a summary on each HTML file that contains any occurrences of any words found. Each file represented in the summary include a set containing words, with the words occurrence count and similarity score. A new routine will go over these summaries and calculate a score for each HTML page. The total HTML score calculation is oversimplified in this pseudocode:

```
for HTML in summary:
    for chosen_word in HTML as W
        HTML.score += W.count * W.similarity
HTML_ranks = summary.HTML sort by scores
```

The very same process goes for the JSON structure created by the XML Evaluator. Except that the number of times each word occurs are not used in the calculation of the end score.

```
for Schema in summary:
    for chosen_word in Schema as W
        Schema.score += W.similarity
Schema_ranks = summary.Schema sort by scores
```

The two ranking lists are ultimately combined and sorted by their total score. These results are translated to DOT and then to a pdf that is presented to the end user. An example on such an display can be seen in **Appendix B.1** (the example does not include hyperlinks)

[2]https://www.datamuse.com/api/

6 Experiments

This chapter explains how tests of the system are planned out and how these are executed. Results from these experiments will be examined and discussed throughout this chapter as well. The goal of this research is to get an understanding of how well or poorly the implemented program from **Chapter 5** serves as a solution to the **Problem statement 1.1**.

6.1 Data Sources

A stable environment is required for a meaningful testing phase. So three documentation domains were chosen to keep the testing within bounds. To recap from **Chapter 5.1.2** where test domains were mentioned. Three domains were used for testing and these are:

Gettyimages	- Documentation for a stock image REST API by Getty images.
Apimet	- Documentation for data collected by the Meteorological Institute.
Datamuse	- Documentation for an API for word-finding query engine.

All the testing data that is used and every interaction comes from or are done with these three. Naturally, the contents from these domains will not be identical and **Table 6.1** shows the relevant contents for the domains. This includes the total amount of websites that the crawler needs to crawl, the amount of URIs that return as HTML files and XML Schema.

After crawling all these domains, indexing every HTML and evaluating every XML Schema do the system create a library of data that amount to 14.73 MB. While out of a total of 170 HTML files will the indexer claim to index 172 pages. This number is larger than the total number of files to index so this is clearly a bug. This could imply that 2 pages get indexed twice which should favor them since the

API	Sites	HTML Files	XML Schema
Gettyimages	150	92	0
Apimet	270	75	28
Datamuse	10	3	0

Table 6.1: Test domain statistics.

occurrence-rate for the words gets doubled. What triggers this, when or why is unknown.

6.2 Tests

Experiments carried out are labeled with three different tags. Firstly A_n and H_n refers to tests that are aimed at checking the soundness of the solution. In other words, these tests are devised to check if the different functionalities create results that useful for any user or any subsequent module in the chain. A_n are reserved for tests on data acquisition i.e. the crawler. While H_n are reserved for tests on the Key Phrase Database and the API recommendations. The third set of tests are labeled with the T_n tag. These tests are aimed at checking out the performance of the various components of the solution.

- **Experiment *A_1*** - Comparison of hierarchical graphs to correspondent domains.
  Goal: Validate that web pages are nested the way the graph display them as. This is done to make sure that the crawler actually crawls through the domains in the intended fashion. It is also aimed at making sure that the physical hierarchy gets nested in a proper way. The problem with a poorly nested physical hierarchy is that the XML Schema might get connected to unintended HTML files. Which makes the scores for the API recommendations all wrong.
  Experimental setup: *Run crawler as deployer to create the URI hierarchy graphs. Then manually iterate the domain and compare if the endpoints match.*

- **Experiment *A_2*** - Check that the HTML, XSD and XML files gathered are valid and without errors.
  Goal: Validate that the crawler handles response bodies correctly and stores files with the correct file extensions. This is done to make sure that the data that gets indexed or evaluated are structured in a way that can create a sound Key Phrase Database.

Experimental setup: *First run crawler without active indexation. Then run a script that iterates every HTML in the physical hierarchy which is validated by tidylib.*[1] *The same is done for XML files by parsing them with lxml.*[2]

- **Experiment *H_1*** - Validation of Key Phrase Database.
  Goal: Makes sure that data within the Indexation Table and the Schema Table conform to a existing data format. This is done to make score calculations more predictable.
  Experimental setup: *Use of the JSON Schema module to compare the Indexation Table to the JSON Schema file that can be seen in **Appendix C.4**. The same process is done for the results from the XML Schema evaluation, JSON Schema used for this purpose can be seen in **Appendix C.5**.*

- **Experiment *H_2*** - Locate search terms in API recommendations.
  Goal: Make sure that the recommended APIs contain the search words applied by the user. A failure on this part reduces the solutions credibility and overall usefulness.
  Experimental setup: *Manually visit HTML and XML Schema of the top recommended APIs to count the occurrences of search words.*

- **Experiment *T_1*** - Profile function time.
  Goal: Find out which functions that use the most amount of execution time. This is useful for finding bottlenecks in the service and to locate areas that can be optimized.
  Experimental setup: *Running the service as a deployer with pythons cProfile*[3] *module active to get deterministic profiling on the crawler, indexer and schema evaluator.*

- **Experiment *T_2*** - Check if components are IO or CPU bound.
  Goal: Find out what if the components are IO(input/output) bound or CPU bound. This information tell a great deal on where optimizations can be made.
  Experimental setup: *Use the psutil*[4] *module to get CPU load percents for the running components.*

- **Experiment *T_3*** - Profile memory use of segments.
  Goal: Test the system for memory usage.
  Try to locate memory leaks and check to see how the memory consumption

[1]http://www.html-tidy.org/
[2]https://pypi.python.org/pypi/lxml
[3]https://docs.python.org/3.6/library/profile.html
[4]https://pypi.python.org/pypi/psutil

behaves during execution. Scalability will also be discussed in regard to memory consumption.
Experimental setup: *Run each stage of the computational-chain with the memory_profiler active.*[5]

- **Experiment T_4** - Crawler File Sizes.
  Goal: Get an understanding on how the crawler behave when the data sizes changes. How handling speed of the solution trends based on a range of factors can be used tell what causes slowdown.
  Experimental setup: *Logging time measurement between API responses, response body size and Indexation Table size.*

- **Experiment T_5** - Indexer File Sizes.
  Goal: Get an understanding on how the indexer behave when the data sizes changes. **Experimental setup:** *Logging time spans between each indexed HTML file, its size and the Indexation Tables size.*

Tags	**Goals**: Why the experiment was conducted.
Experiment A_1	Validate domain crawl restrictions and hierarchy nesting.
Experiment A_2	Make sure gathered data is fit for evaluation.
Experiment H_1	Make sure the Key Phrase Database gets structured as intended.
Experiment H_2	Check if API recommendations contain the search words.
Experiment T_1	Find out which functions that use the most execution time.
Experiment T_2	Identify H/W[6] improvements that would increase performance.
Experiment T_3	Check for memory leaks and trending memory expense.
Experiment T_4	Identify reasons for increased crawl time.
Experiment T_5	Identify reasons for increased indexation time.

Table 6.2: Short summary on test goals.

6.3 Results

6.3.1 Experiment - A_1

This test was done by manual traversal of the domains in question while simultaneously comparing the paths to hierarchical graphs. An example of such a graph

[5]https://pypi.python.org/pypi/memory_profiler

can be seen in **Appendix A.2**. The links embedded in the graphs directs the user to the correct locations. Some branches of the graph direct to locations that are not visible as hyperlinks in the websites visited. This is because the crawler finds links that are written as comments in the HTML code or that are in other ways hidden from the user. No link found in any graph direct any user to an outside domain.

6.3.2 Experiment - A_2

Running every HTML file stored on disk through the tidylib module yielded the collected results seen in **Table 6.3**. Through a total of 170 HTML were there 1318 warning and 0 errors. This means that every HTML file was written in an intelligible way and that the data format is predictable. The high amount of warnings are less of a concern as they pose more as a problem for the actual developers of the HTML files in what they really try to convey.

	Warnings	Errors
HTML	1318	0
XML	0	0

Table 6.3: Summary of HTML validation.

Every XML and XSD file stored on the disk was run through the lxml module to find errors in the files. Through a total of 76 files were there no errors. This means that the files read by the XML Schema Evaluator are fit for evaluation.

6.3.3 Experiment - H_1

The JSON Schema module is designed to throw an exception whenever an error between the JSON document and the JSON schema occur. No such exceptions were thrown while testing the Indexation Table. Additionally, jsonschemalint[7] were used to make sure that the schema used for validation were written correctly. The linter lets users check if their schema follows the JSON Schema conventions and can be used to try it out in action against JSON documents.

The Indexation Table was inserted into the linters document section to find out how lenient the schema really is. This was done by changing a wide variety of

[7]https://jsonschemalint.com

changes to value typings, key names and overall structure of the document. The conclusion is that the schema is really strict and that leads to the conclusion that the Indexation Table gets constructed in the designed way.

The same process was replicated for the output from the XML Schema evaluation and no exceptions were thrown. Therefore the same conclusion was drawn from this test as well.

6.3.4 Experiment - H_2

The API documentation that was investigated in this experiment comes from the top 4 recommended APIs from a run of the application as an end user. These results can be seen in **Appendix B.1**. Data seen in **Table 6.4** comes from manually looking up the keywords in both the HTML file and the XML Schema of each API documentation listed. Words from the HTML files were counted while words from the XML Schema were checked for presence.

	HTML			XML Schema		
API (Hyperlinks)	wind	latitude	latitudes	latitude	Wind	Score
locationforecast	3	1	0	True	True	7.000
extremeswwc	0	0	0	True	True	6.248
upperwindweather	4	0	0	False	False	5.000
spotwind	0	0	0	False	False	4.417

Table 6.4: Word validation on recommended APIs.

It was clear when looking up the words in the files that the lookup is very strict. For instance will "latitude," as a substring be overlooked when trying to find "latitude". On the other hand, will the version with a comma in the end be presented as a possible inclusion that the user might choose to include. Furthermore, a comparison between the words "latitude" and "latitude," creates a similarity score of 0.941 while "wind" and "wind," creates a similarity score of 0.889. So the calculation of similarity score keeps the service from reliably taking decisions such as this on its own. The proposed words "WindSpeed" and "spotWind" were included that pushed the spotwind API onto the list while it contained none of the strict search words.

6.3.5 Experiment - T_1

note: Tables shown in this experiment are simplified versions of the profiler output.

Table 6.5 and **Table 6.6** show execution time of the crawler when it crawled the three test domains. The tables show profiled data from the use of the python cprofile module.[8] These tables contain a list of functions and their total runtime seen as Cumtime. Functions displayed in this list are nested together where the top function is the root function. The first function is in both cases the content switch that feeds the crawler the URI seeds. Its called one time in both tables. Following functions are child functions of a function located above in the listings. The next function can be seen to run 3 times, this is because it is the main function of the crawler that runs once for each domain. At the bottom is the communicator that uses all of its time displayed as HTTP communication. As seen in the tables the communicator attempts to communicate with 368 endpoints.

Table 6.5 was profiled with the inclusion of the JSON Schema feature, while **Table 6.6** does not include the use of this utility. The inclusion of the JSON Schema evaluator required a total execution time 157 seconds while it only took 48 seconds to run the same set of functions excluding the evaluator. Out of the 157 seconds were 96 seconds spent on validating schema. This left 30.5% of the execution time to the rest of the crawler. A bottleneck this large led to the exclusion of the JSON Schema validator. A compromise could have been to reduce the frequency of the validations but the validator was removed completely. The curious this about removing the validator is that the communication time also went down.

This could have been affected by alterations in memory consumption. The total number of function calls went down to 2.3% of the original. This could imply a large amount of stack tracing with many memory allocations and garbage collections. Pythons garbage collection is triggered by a check that activates every time an object gets allocated, referenced or dereferenced. Which should mean that the GC(garbage collection) is run often in the case of the validator. Another thing to note is that the GC runs on the same thread as the rest of the python program. So memory should not be affected as much if there are no memory leaks. The thing that could effect the communicator in regard to memory is if the memory used for the communicator gets moved from one cache to another. All of this is mere speculation and is not tested in any way.

Table 6.7 shows 20 consecutive runs of the indexer without the crawler involved,

[8]https://docs.python.org/3.6/library/profile.html

Function Calls: 96311214
Primitive Calls: 89966533
Ordered by: Cumulative Time, 157.731 seconds

Ncalls	Tottime	Percall	Cumtime	Percall	Function Description
1	0.001	0.001	157.742	157.742	Crawls Domain List
3	0.018	0.006	153.862	51.287	Crawls Single Domain
161	0.018	0.000	103.543	0.643	UPDA Indexation Table
79	0.230	0.003	102.132	1.293	UPDA HTMLindex List
78	0.078	0.001	101.809	1.305	Indexes HTML Code
155	0.003	0.000	96.808	0.625	Validates Index Table
368	0.003	0.000	48.231	0.131	HTTP Communication

Table 6.5: cProfile of indexed crawl with JSON Schema validation active.

Function Calls: 2224440
Primitive Calls: 2195275
Ordered by: Cumulative Time, 48.338 seconds

Ncalls	Tottime	Percall	Cumtime	Percall	Function Description
1	0.001	0.001	48.346	48.346	Crawls Domain List
3	0.016	0.005	45.866	15.289	Crawls Single Domain
368	0.003	0.000	39.479	0.107	Reacts to HTML Response
368	0.002	0.000	35.937	0.098	HTTP Communication

Table 6.6: cProfile of indexed crawl without JSON Schema validation active.

this is listed as the Content Switches 20 Ncalls (number of times the function is called). The content switch set up to destroy the Indexation Table for each run so the indexer can create a fresh table each time. The indexer will it examine 172 HTML files for each of these runs, which are the 3440 calls divided by 20. The JSON Writer will run twice for each file that the indexer indexes. It runs twice since it needs to update the Indexation Table and the table that keep track on indexed files.

Approximately 67% of the indexation time is spent on the JSON Writer and Reader. This leaves only about a third of the execution time to the indexation itself. Keeping the Indexation Table in memory over several indexation sessions could be an effective way to reduce this bottleneck. And instead of writing it to disk when the crawl ended or after a set amount of indexations. If this were to be implemented, then it would be really important to handle the table of indexed files the same way.

Table 6.8 shows 20 consecutive runs of the Schema Evaluator. Out of each evaluated schema were approximately 55% of the time used for traversing through the

Function Calls: 85095278 Primitive Calls: 85069419 Ordered by: Cumulative Time, 395.957 seconds					
Ncalls	Tottime	Percall	Cumtime	Percall	Function Description
20	17.230	0.861	390.564	19.528	Content Switch
3440	3.964	0.001	368.108	0.107	Index Page
6920	0.203	0.000	145.592	0.021	Write JSON
6860	0.131	0.000	101.918	0.015	Read JSON

Table 6.7: cProfile of Indexing NO crawl.

file to extract the data. The function that does this is written recursively so an attempt at reducing this time was committed. The attempt was to rewrite it iteratively, but the execution time did not go down, surprisingly. The fourth function in the list is hard to gauge as it includes HTTP communication.

Function Calls: 24792987 Primitive Calls: 21221231 Ordered by: Cumulative Time, 216.152 seconds					
Ncalls	Tottime	Percall	Cumtime	Percall	Function Description
20	0.078	0.004	216.152	10.808	Content Switch
580	0.733	0.001	163.716	0.282	Switch XML Schema
900	138.556	0.000	141.550	0.157	Traverse XML
580	1.006	0.001	38.567	0.066	Get XML Schema from source
1440	26.472	0.018	30.043	0.021	Shave Text
900	0.017	0.000	21.240	0.024	ElementTree parse

Table 6.8: cProfile of XML Schema evaluation NO crawl.

6.3.6 Experiment - T_2

Figure 6.1 shows the CPU load on the crawler while running with active indexing. Again this is run over three API documentation domains and the figure shows an average of 20 runs on all of them. The y-axis shows the processing load of the thread relative to one core. While the x-axis shows the CPU time of the process, not to be confused with execution time. CPU time is the amount of time for which a CPU was used for processing instructions of a thread. And **Table 6.9** shows statistical information in the same data.

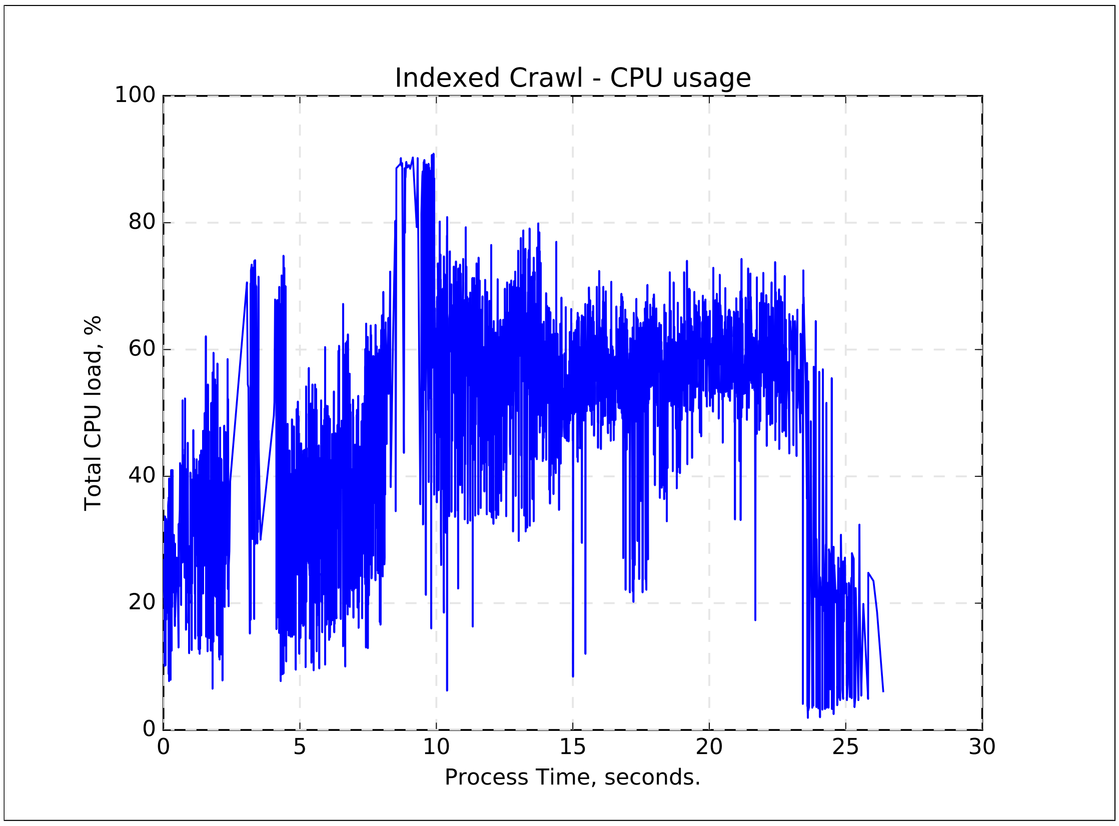


Figure 6.1: CPU drain during indexed crawl over 3 domains.

We can see that the actual load on the CPU is not really near 100% which suggest that the crawler is IO bound. This statement can be supplemented by looking at the communication time from **Table 6.5** where approximately 70% of the runtime is used on HTTP communication. A solution to this could be to parallelize the code and run HTTP pipelining [19] to reduce the communication time. If the code were to be parallelized, then multiple APIs could also be crawled at the same time. This could reduce any bottlenecks set in place by the API documentation sites.

Another room for improvement is to avoid using urllib that is currently used in this implementation. The problem with urllib is that it doesn't support persistent connections. By changing it to http.client that support persistent connections trough legacy http/1.0 keep-alive [20].

Figure 6.3 shows the average CPU load on the indexer over a set of 20 runs on HTML files collected from all three test domains. The y-axis shows the processing load of the thread relative to one core. While the x-axis shows the CPU time of the

	CPU	Memory
Mean	47.339	91.923
Standard Deviation	16.882	5.138
Variance	285.022	26.409
Minimum	1.9	3.781
Maximum	90.9	97.714

Table 6.9: Statistical description of data from indexed crawl

process, not to be confused with execution time. And **Table 6.10** show statistical information in the same data. Since the CPU load is so high can it be assumed that the indexer is CPU bound. This is a little surprising as **Table 6.7** show that 67% of the computation time is spent on the JSON write and read functions.

This could make sense if the functions named "loads" and "dumps" from the JSON module of python use a long time translating the python objects to JSON code. An objous alternative to optimizing the code is to rewrite some code to make it concurrent and then run indexation of several pages in parallel.

A new module was found that might help the program with the slow runtime of the JSON translations. This module is named ujson[9] and is written for python and claim to be a faster JSON encoder than the original. A minor experiment that test ujson compared to json to see if it actually could help this system. The test uses the formula 6.1 from **Figure 6.2** that iterates through each time stamp from the dataset described in the third stage of **Experiment - T_5** which can be seen in **Figure 6.11**. The formula contains the sum mean time used for indexation for each size variation of the Indexation Table. The result of the formula is the average time used to index one megabyte of raw HTML data. The regular JSON module used 0.107 seconds in average and ujson used 0.084 or 78.5% of the original. This is deemed a good improvemnt so the ujson module was left in. Upcoming data was collected before the inclusion of the usjon module.

$$Speed = \sum_{x=0}^{172} \frac{\bar{t}_{\mathrm{x}}}{s_{\mathrm{x}}} \cdot 2^{20} \tag{6.1}$$

Figure 6.2: Equation 6.1: Speed calculation of json and usjon.

Note: CPU load is seen to go above 100% which means that some imported module uses multithreading.

[9]https://pypi.python.org/pypi/ujson

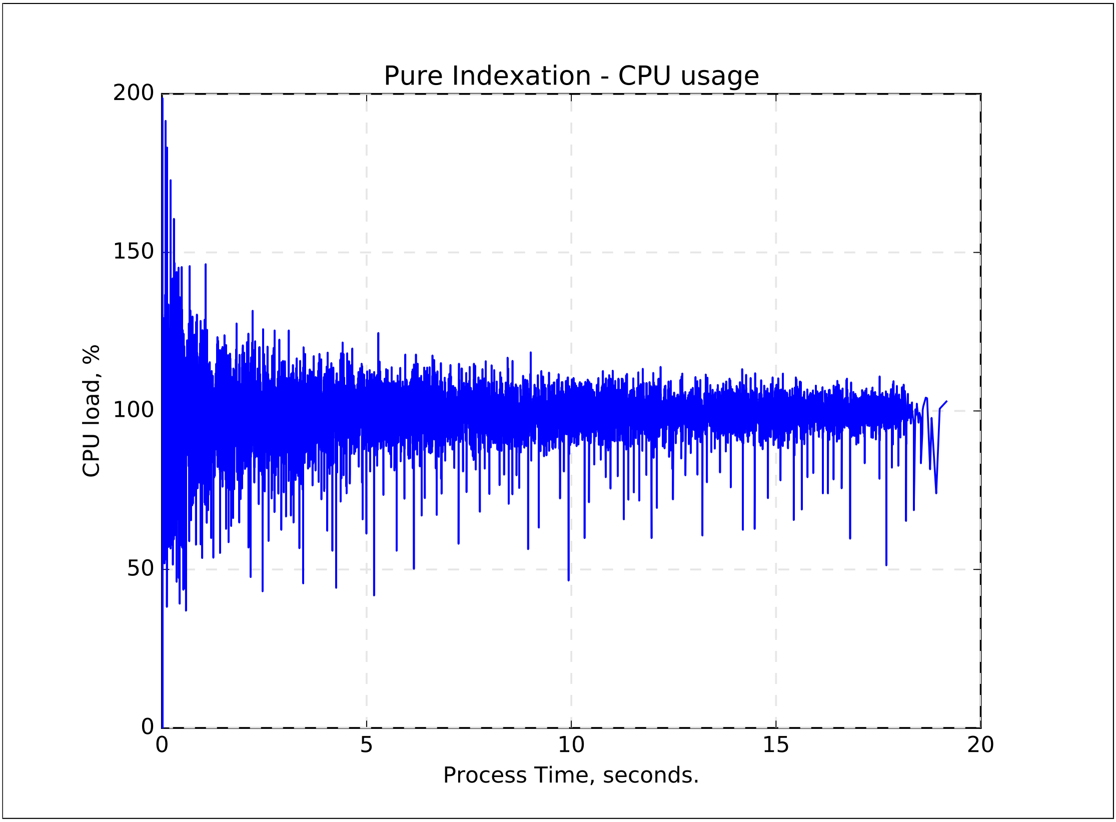


Figure 6.3: CPU drain during pure indexation of HTML files from 3 domains.

6.3.7 Experiment - T_3

Each of the tested sections deals with files that will increase in size over time as the crawler find new data to examine. Like for instance will the Indexation Table increase in size as new HTML files get discovered. Because of this is it expected that the modules that read these files into memory, will gain a great deal in memory consumption as time goes by. Memory consupconsumptiontion to filesize comparison test is however not conducted.

Experiment - T_3 Crawler

The first test carried out in this experiment was on the crawler with active indexing during the crawl. Data collected for this test was done by repeatedly indexing the three domains 20 times each while monitoring the memory consumption. The

	CPU	Memory
Mean	99.114	86.242
Standard Deviation	9.283	7.292
Variance	86.181	53.174
Minimum	0.0	9.300
Maximum	198.7	92.832

Table 6.10: Statistical description of data from pure indexation

results can be seen in **Figure 6.4** with the statistical information listed in **Table 6.9**. A cleanup of crawled data written to disk was executed between each new iteration of the three domains. This made each iteration of the domains in theory identical. During the first iteration was the memory allocated to the crawler around 75 to 80 Mb while entering the third iteration was the memory consumption up to around 90. Its expected that the first iteration should increase the memory consumption but the drastic increase from the first iteration to the second is somewhat surprising. The incremental step of memory consumption between iteration one and two are larger than the increase between iteration two and nineteen. Furthermore, the memory seems to creep upwards from when the point when it reaches 90 Mb. This could lead to a problem when running over a longer period of time. The way the Indexation Table is structured to be written also poses a problem down the line. As its increase in size also will effect the memory consumption of the crawler when indexation is active. This leads to the conclusion that the service has a scalability problem.

Experiment - T_3 Indexer

The next test of this experiment was done on the indexer alone, the graph on the data collected can be seen in **Figure 6.5**. 20 iterations over the three domains that in total contain 172 HTML files, were conducted as a basis for the collection of data on this test as well. Between each iteration can it be seen that the memory goes down so the memory used occilates between 83 to 93 Mb. The indexer does not appear to have problems with memory leaks as the level of memory consumption seem stable.

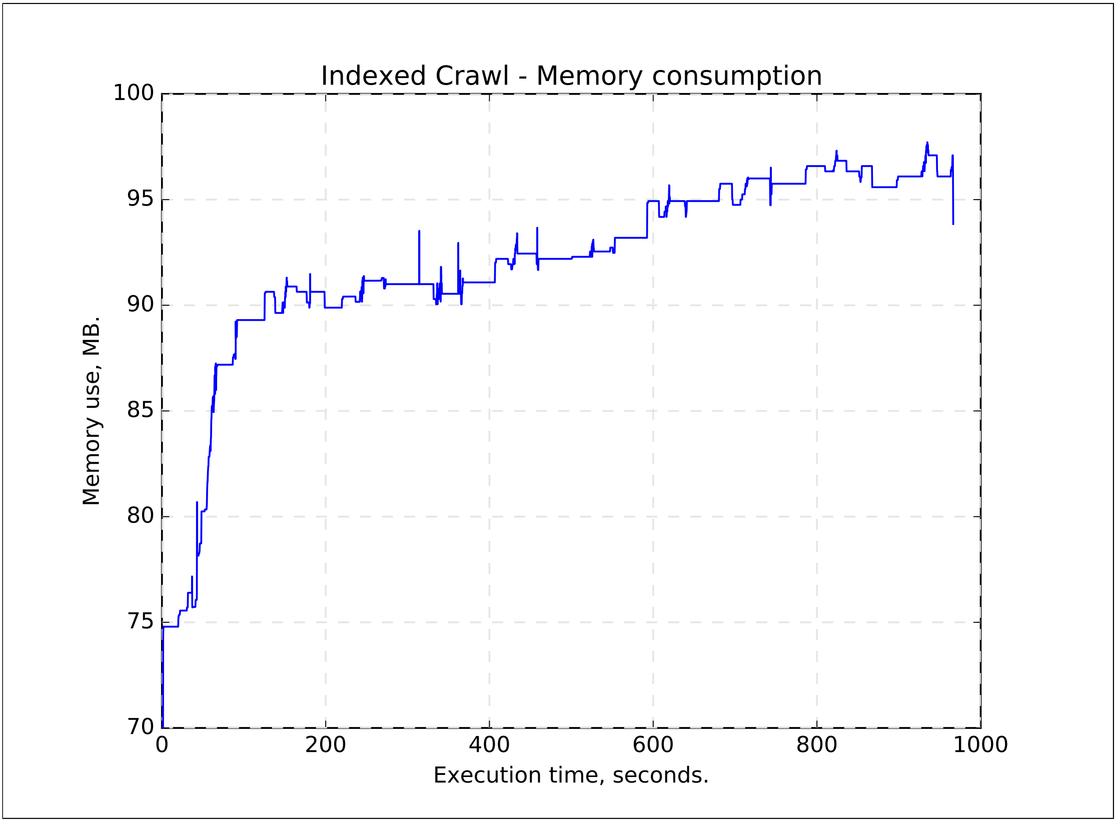


Figure 6.4: Memory consumption during indexed crawl over multiple domains.

Experiment - T_3 Schema Evaluator

The last step of experiment T_3 was to test the schema evaluator. This was done over 28 XML Schema all collected from api.met.no, data gathered was intended to be from all three domains but api.met is the only one with XML Schema. Like the other tests, this one has run over 20 iterations and the results can be seen in **Figure 6.6**. Running schema evaluation in between other tasks seems to be working without any real memory leak problems. The middle section has a longer execution time than the rest. This happened because other processes in the background were active.

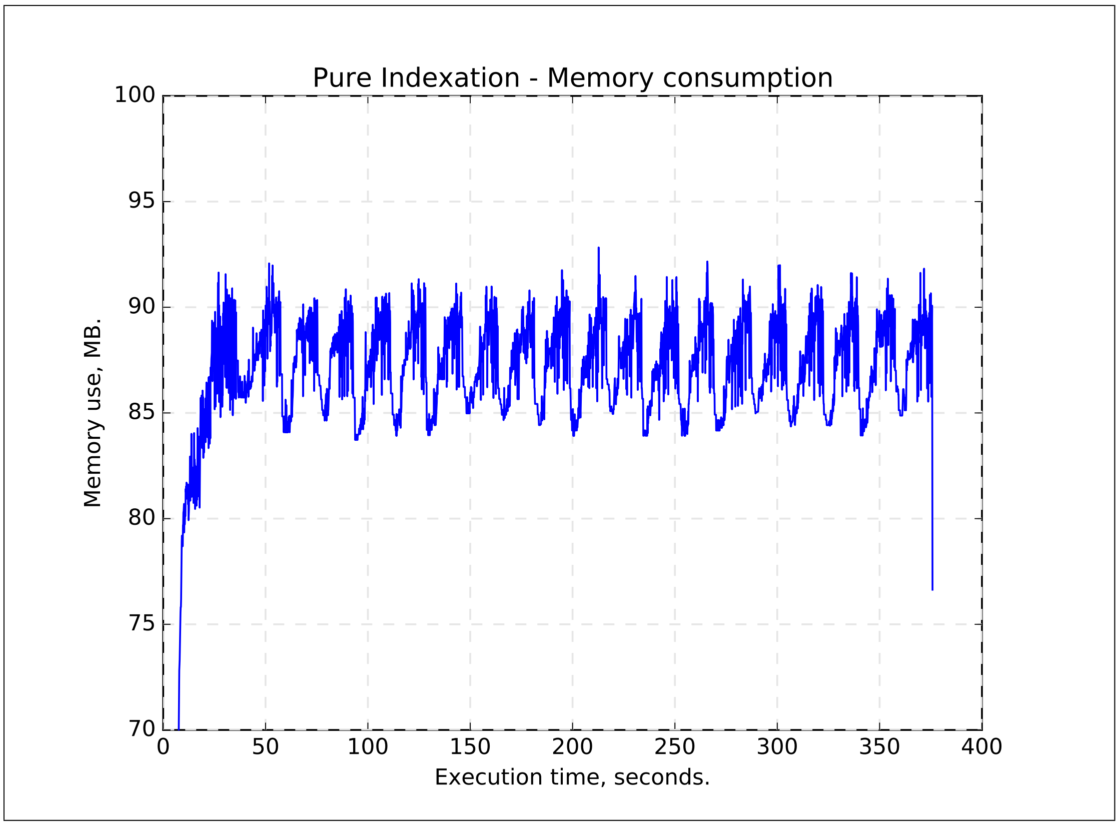


Figure 6.5: Memory consumption during pure HTML indexation.

6.3.8 Experiment - T_4

The first section of this test is to see how the size of the Indexation Table effects the crawl time during crawl with active indexing. This was done by timing the execution time on the crawler between each received HTTP response and storing the time value together with the size of the Indexation Table at that moment. Responses containing filetypes other than HTML were not included, so every entry in the plot have combined indexation and crawl time. Additionally, the response time from the HTTP request was removed as well, so the execution time of the crawler and indexer is the only process measured. The crawler iterated through 20 cycles of three domain crawls in order to get the data for this experiment. The mean of time span for each size variation can be seen in the graph in **Figure 6.7**. where the x-axis show the size of the Indexation Table and the y-axis show the time it took to crawl one website. Some data points are left out of the mentioned Figure as they made the graph a lot harder to view. These can be seen in **Appendix C.2** where

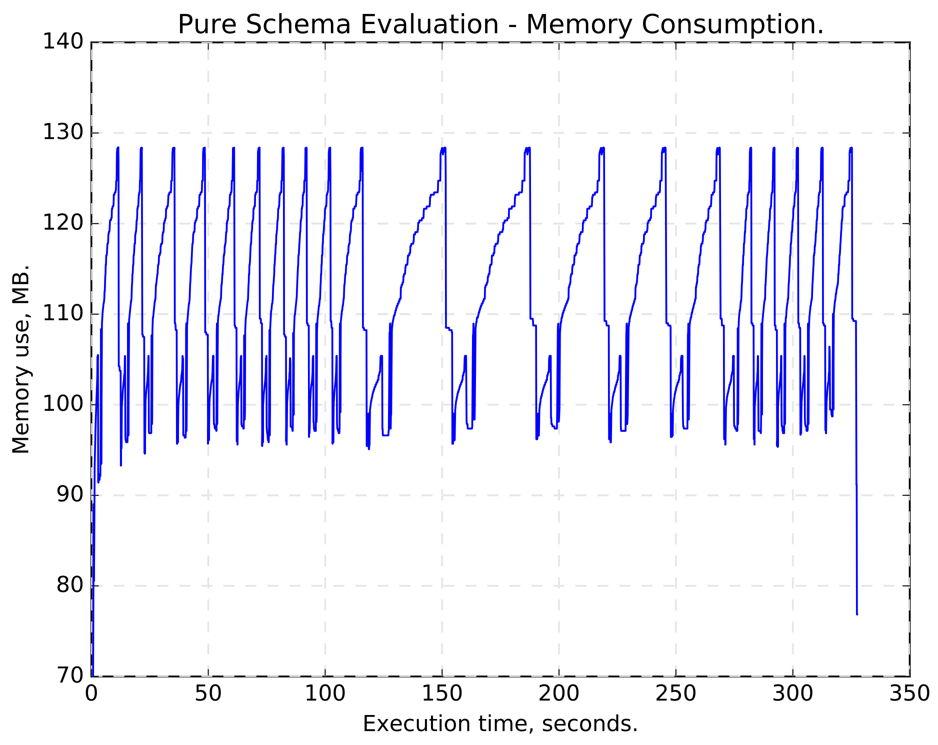


Figure 6.6: Memory consumption during schema evaluation.

the graph with the outliers is left in.

The graph from the **Figure 6.7** can be argued to be somewhat linear with some outliers here and there. An increase in executiontime is realy expected as the Indexation Table is written to disk and read between each indexation. Which should take longer time based on the size of the Indexation Table. An exponential growth had been way worse compared to the linear natrue of the system as we see here. How the crawler act when the table grows outside the range of the Table is unknown but this small sample range suggest that it might keep a linear progression.

The outliers are however troubling as they make the flow of the service somewhat irregular. Each of these outliers come from the same domain and the sites were they come from are also listed in the Appendix. The nice thing is that the outliers allways came from the same sources. Exactly what the problem is, is not known but one thought was that it could be the size of the return package that effected the crawl time. Which naturally could be the case, so the next segment is on how the return file size effects the crawler. The outliers will be further discussed there.

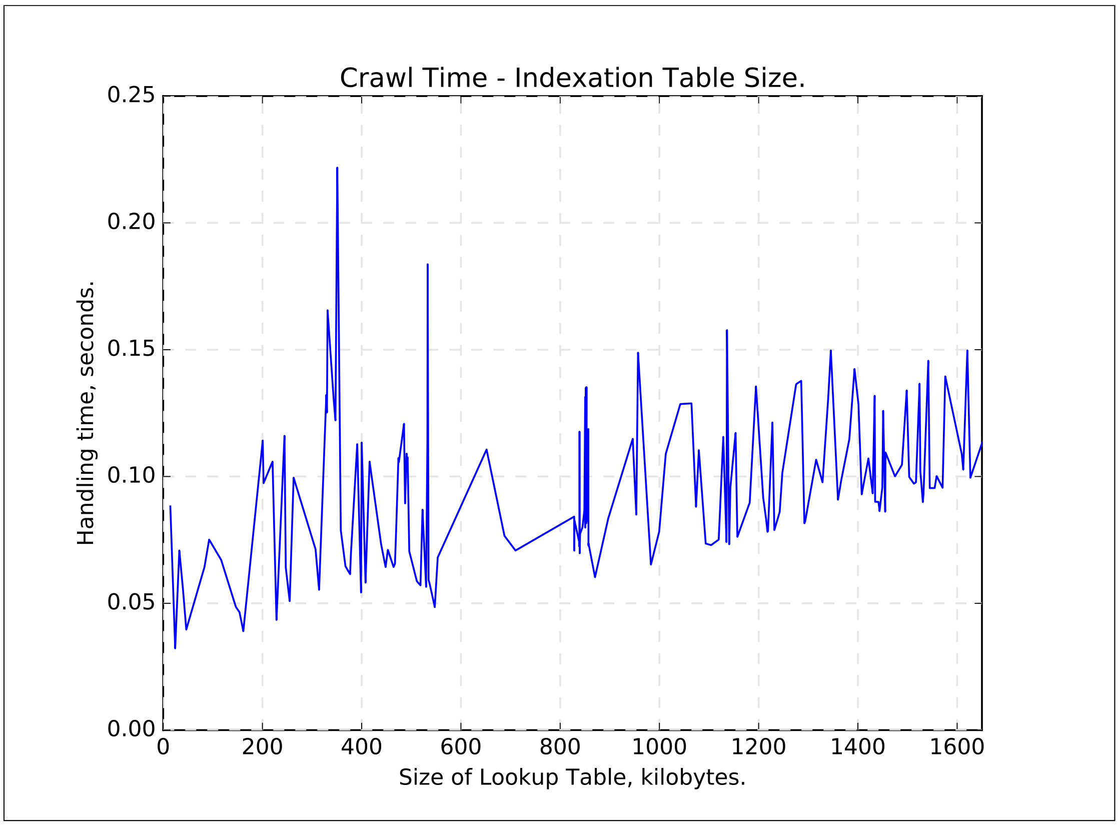


Figure 6.7: Runtime of indexed crawl based on Indexation Table size.

The same process for datacollection for the second segment of this experiment was conducted the same way as the first but with some small changes. Active indexing was turned off and the size of the returned HTML file were recorded instead of the size of the Indexation Table. The data can be seen in the graph in figure **Figure 6.8** where the x-axis shows the size of the HTML file received in kilobytes and the y-axis show the time used to handle everything from when the previous packeage was received. Recorded time used on the HTTP communication were subtracted for each packet received. Some outlier were removed from this graph as well and can be seen in **Appendix C.3** where graph have the outliers included.

The outliers removed from this graph came from the same sources that got removed from **Figure 6.7** reasently discussed. So the packet size was not the reason for the exatra delay for the outliers. And no test have been successfuly used to identify the reason for the larger outliers.

Handle time increases as the response body increases as seen in **Figure 6.8**. How the progression act can be hard to extract from the graph as it contains few entries with higher response volumes.

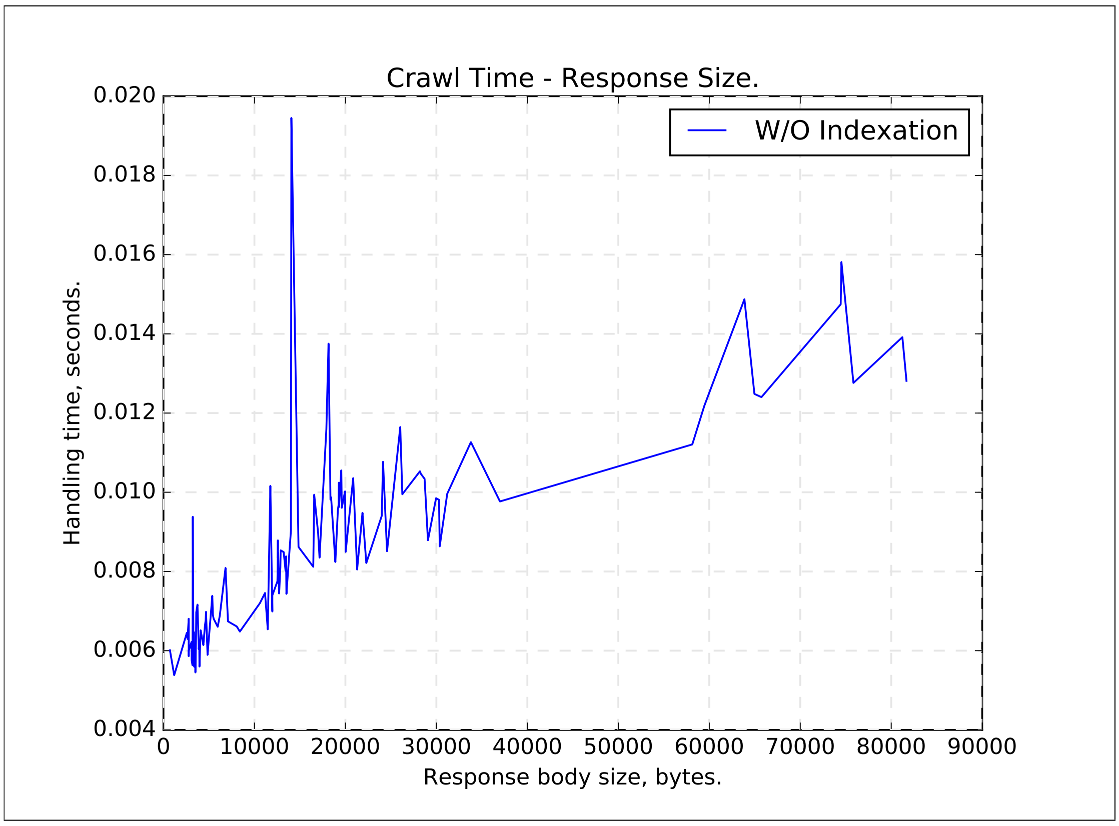


Figure 6.8: Runtime of pure crawl based on HTTP response size.

The third section of this experiment was to see how the total execution time effected crawl-time of single websites. The experiment was conducted by running the crawler without active indexing through the three domains 10 times while recording the total process time with the time between each crawled endpoint. The results can be seen in **Figure 6.9** where the x-axis shows the total process time at that very moment and the y-axis shows the time between each file response from any endpoint. Note that the x-axis actually shows process time and not execution time, which is way higher. As seen in the Figure does the actual time between packets go down. This is because of a change in domain crawl during the total crawl session. The mean of each tripple domain cycle cas calculated and cen be seen as the red dots in the graph. The statistics on these dots eas calculates as well, with a mean of 0.008, a standard deviation of 0.0002 and a variance of $4.07*10^{-08}$. This makes the standard deviation around 2.4% of the mean which does not makes little ground for any increase. The increase might be there but the data can not support that claim.

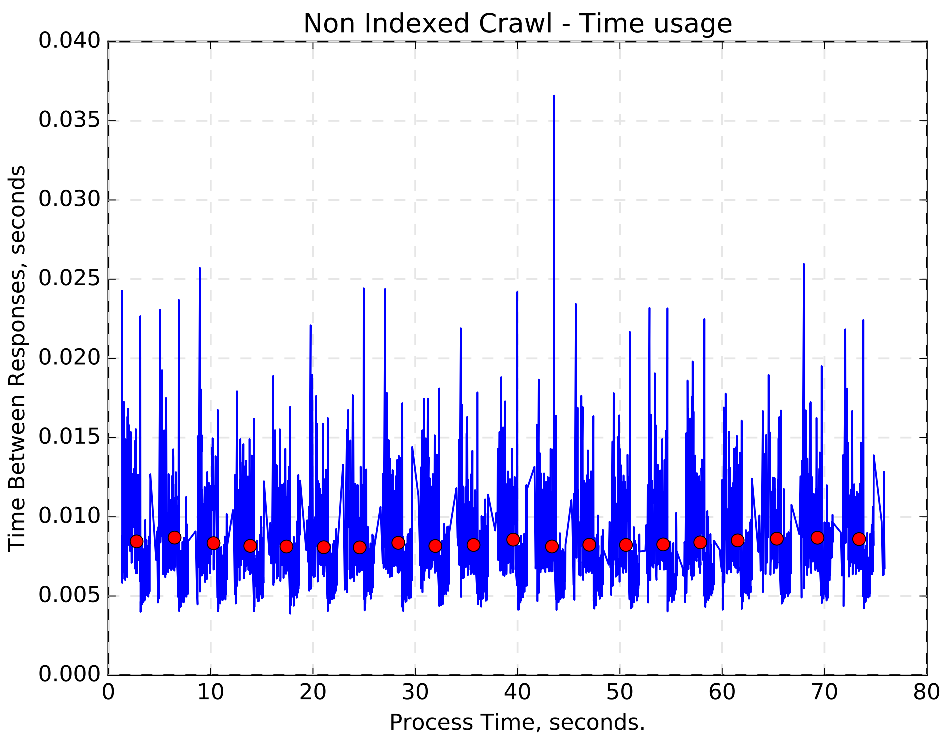


Figure 6.9: Runtime of pure crawl based on total runtime.

6.3.9 Experiment - T_5

This experiment is aimed at finding behavior patterns in the Indexer when changing the size of IO components. First a test on how the HTML files affect the indexer was conducted. This was done by running the indexer over HTML files from the three domains 20 times and letting it index everything each time. The time span between each indexed HTML file was recorded along with the size of the actual file. In hindsight should the Indexation Table be altered in between each iteration so a fixed table could be used for the experiment. Some additional variance is therefore introduced since the size of the Indexation Table changed along with the execution of the indexation. On the other hand, the order of HTML files was randomized between each iteration so there is some merit to the graph, which can be seen in

Figure 6.10. The x-axis shows the size of the HTML files while the y-axis shows the time it took to index that file in seconds. To be open about it, this graph does not yield enough stable information to make any sound statement on how the indexer react to the increase in HTML sizes. The variance is to high and the lack of data in higher file sizes makes it inconclusive.

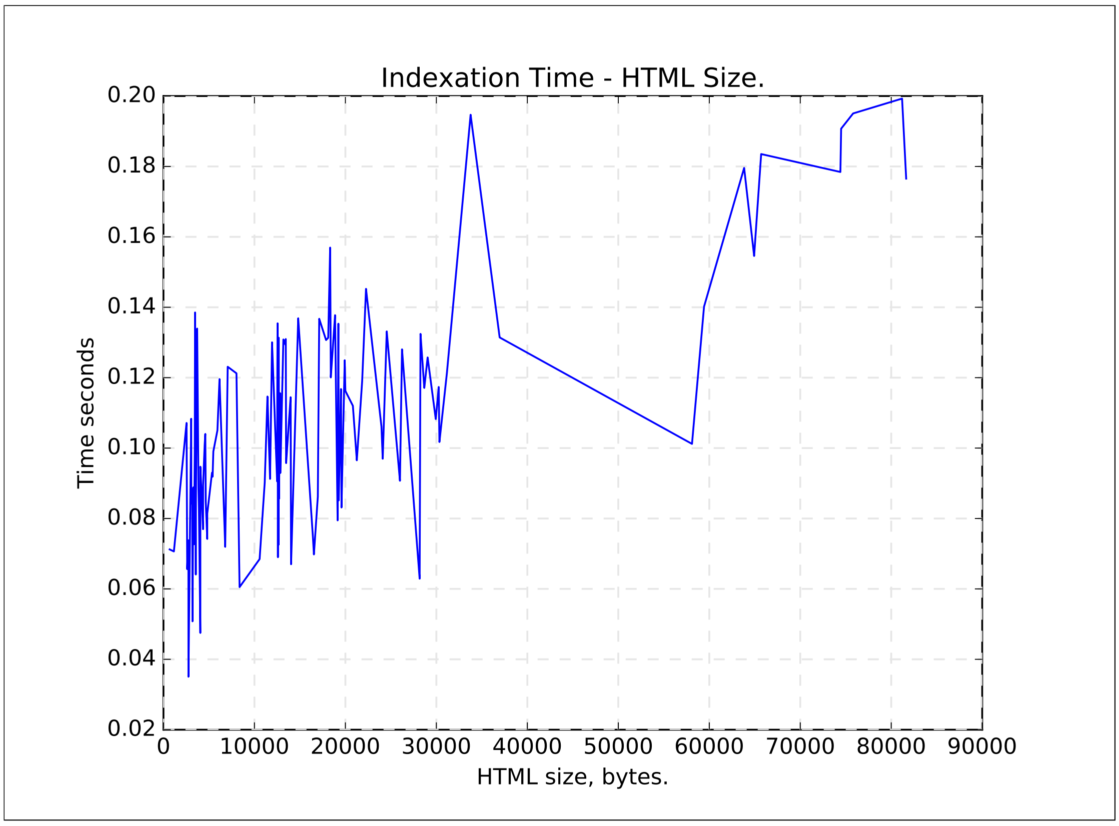


Figure 6.10: Runtime of pure indexation based on HTML file size.

The second section of this test is to see how the size of the Indexation Table effects the indexation time of the Indexer. This was done by timing the execution time on the indexation of one HTML file and storing the time value together with the size of the Indexation Table at the moment of completion. This was done for every HTML file iteratively on all files collected from the three domains. This process was done 20 times where the HTML files that got indexed were indexed in a random order for each time. This was done to keep the influence of the current HTML file size from bearing a too large imprint on the test results. Results from the test can be seen in **Figure 6.11** where the x-axis shows the size of the Indexation Table and the y-axis show the time it took to index one random HTML file.

The graph from the **Figure 6.11** can be argued to be somewhat linear. An increase in execution time is really expected as the Indexation Table is written to disk and read between each indexation. Which should take longer time based on the size of the Indexation Table. An exponential growth had been way worse compared to the linear nature of the system as we see here. How the indexer act when the table grows outside the range of the Table is unknown but this small sample range suggest that it might keep a linear progression.

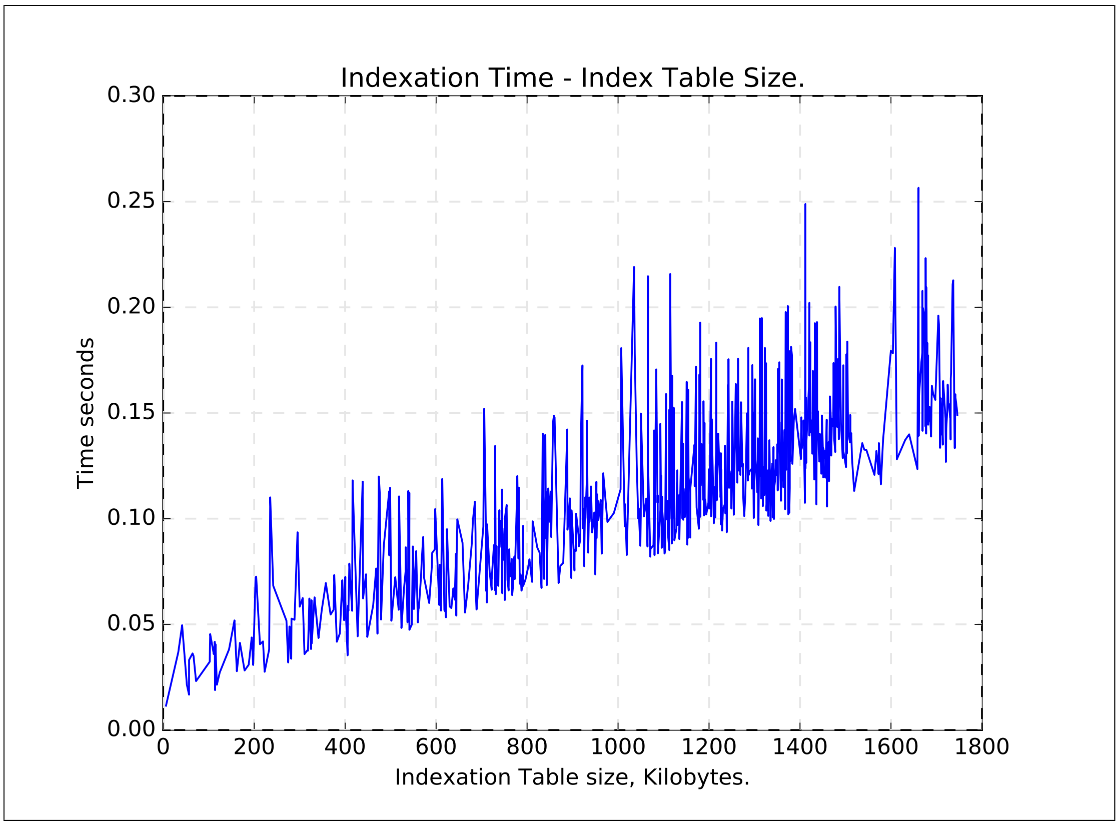


Figure 6.11: Runtime of pure indexation based on Indexation Table size.

7 Discussion

This chapter will reflect on the process of working on this thesis and how the resulting end product stands in comparison to the **Problem Statement 1.1**: *Design a set of automatic overview mechanics presenting resource location and relevance on key phrases related to open APIs as a mean to reduce development delay due to open API documentation exploration.* This will include discussion on how the project evolved in regard to analysis of the project as a whole.

7.1 Project Evaluation

Let's start out with a brief summary of what the solution is comprised out of. The key set of features outlined in this thesis have all had one greater purpose, to reduce the development delay for users that need to find APIs that fit their design. A proposed solution has been to create a crawler that creates an overview on APIs that can be presented to the user. These APIs were then rearranged based on the user's needs applied as a set of key phrases or words. Descriptions on API input were extracted by normal HTML indexing and output for the API was located through metadata located in schema, all of which are gathered by the crawler. These features were separated into two sections that were termed Data Acquisition and Data Handling which will be discussed now. Discussion on the Data Acquisition will be on how viable the crawler can be in a deployed setting and how well the data gathered suites the purpose of the Data Handler. Discussion on the Data Handler, on the other hand, will analyze how well the Key Phrase Database act in a deployed setting and how well the actual results presented to the user actually help the user in discovery of suitable APIs.

7.1.1 Data Acquisition

Intermediary data collected by the crawler were absolutely crucial in the computational chain of the solution. This is truly the case since everything builds directly upon the discovery of the crawler. Two experiments were devised to verify the data gathered at this stage.

The first experiment; **Experiment *A_1*** tested which websites within a domain the crawler visit and how the representation of the URI hierarchy match the structure of the domain visited. The test showed that the crawler, in fact, does its job as intended in regard to domain traversal. Where the boundaries i.e. restriction to keep traversal within the domain was held.

The validity of the data gathered by the crawler was tested in **Experiment *A_2*** where the HTML files stored to disk were verified to be written in a proper way. By reviewing the results can we see that there are some warning but no errors. This means that the data gathered builds a sound foundation for the indexer and that the indexed data comes from sources that would not direct any user to a dead end. XML files and schema were also tested and the results.

The conclusion on the crawler is that it does as it is designed. Flaws in the design can, on the other hand, be scrutinized as the current implementation is held back in terms of handling time of domains by a verity of reasons. The fact that everything is single threaded and run as one single process makes the entire crawl stop in its entirety whenever the crawler waits on HTTP communication as seen in **Table 6.6** from **Experiment *T_1*** where only 69.5% of the execution time was spent waiting. The implication this has on the system as an entire unit is that the rate at which APIs can get applied to the Indexation Table is reduced. Which reduces the range of APIs that can get represented to the end user. This is of course under the assumption that the crawler will be applied URI seeds frequently enough to run non stop. Another problem that the crawler will face when running without interruption is that has a memory consumption that has an increasing trend as seen in **Figure 6.4** from **Experiment *T_3***. This increase in memory consumption can be a problem but the memory consumption of the crawler without active indexing was never conducted. So there is a chance that the increase in memory consumption is a problem that arises solely from the fact that the Indexation Table takes up room in memory that is included in the graph from **Figure 6.4**. By looking at **Figure 6.9** from **Experiment *T_4*** it would seem that the execution time does not really increase much between domain switches. As seen in the same test the mean between the cycles have a mean of 0.008 and a standard deviation of 0.0002 which makes it seem that the scalability of the crawler as a separate instance is unproblematic.

That assumption is stated as a remark on the crawler separately from the indexer that will be discussed momentarily. On the other hand, if the crawler would be set loose on the entire web. Then it would definitely have a large scalability problem. Additionally, the size of the domains used for testing are somewhat small so the crawler might act differently under larger domain crawls.

One of the biggest drawbacks of the implemented crawler is that it requires manual feeding of URI seeds to the seed list as mentioned in **Section 4.3.1**. This shifts the burden from the end user to the deployer which in praxis should serve to make the end user happy. But the hardship placed on the deployer makes the service unattractive due to the lack of automation on their end. Whats really needed might be an additional smaller and more lightweight crawler that finds domains that contain API documentation that feeds the seed list. Such a feature have been thought of but have not been a focal point in any way.

Much of the data on disk is really unnecessary in an actual deployed setting but were written to disk for development and testing purposes.

7.1.2 Data Handling

Data collected for the crawler as seen in point 1. from **Figure 4.2** in **Chapter 4.4** creates the foundation of future work for the Data Handler as a whole. As seen in the same Figure, two components utilize this information to collectively create the Key Phrase Database (above 5.). The intermittent work sections are the Page Indexer and the Schema Evaluator. These will be discussed as the first section of the data handling process. The second section will discuss point 5 to 7 from the same Figure.

The contribution by the Page Indexer was tested in **Experiment H_1** where the validation of the output structure was conducted. The test came out clean so the Indexation Table gets written as intended. However, the design of the Table creates a large scalability problem down the line for the entire service. It is structured to contain every word, their count and the source of the word all within one file. This single file gets written and read multiple times throughout the indexation phase, during the crawl or otherwise. And the Table will take more and more room in memory and the total disk use will continue to grow as the process goes on since the Table only increases in size. The problem with this is many as the size increase causes the reads and writes to take longer for every expansion of the file. This can be seen clearly in **Figure 6.8** from **Experiment T_4** and even more firmly from **Fig-**

ure 6.11 from **Experiment *T_5***. Ony can only imagine how low the service would run when the memory consumption reaches the threshold of the cache. One possible solution would be to create several files where they only contain a fixed amount of words so the read and write time would go down. The obvious problem with this is that it only postpones the inevitable problem just recently addressed that surely would arise all over again. It would be safe to assume that an entire infrastructure tailor-made for the Indexation Table would be required to turn make the Data Handler scalable.

The same analysis can be applied to the Schema Evaluator that through **Experiment *H_1*** was shown to work as intended. The same scalability problem would, of course, be the same as with the Indexer and would require similar treatment. Fewer tests were made on the Schema Evaluator compared to the Indexer, but they serve a similar purpose and gets the job done in a similar fashion. The Schema Evaluator was made later in development and were written in a way that made it hard to include in the active crawl. This is an oversight that should not have been made as it made testing awkward and that it causes more work for the deployer. Another missed opportunity with the Schema Evaluator is that it has access to the element types of the values it extracts. These should have been coupled with the words to signify what kind of response these fields actually yields.

The second stage of the Data Handler shown in **Figure 4.2** in **Chapter 4.4** as number 4. to 7. will suffer the same scalability problems that the Indexer and Schema Evaluator suffer from. This section is not tested much in terms of performance but it stands to reason that working on the same set of data as the previously mentioned would pose the same set of problems. Specifically, the Indexation Table and the Table from the Evaluator would be read into memory in order to make the API recommendations. The least compelling feature of this section of the Data Handler is that the data required to make the API recommendations are not currently separated from the end user. Which means that the end user actually needs to have the datasets on their own drives to run the Lookup Service. This catch is easy to rectify with small changes but the service in its current form is intended as a proof of concept. One test was conducted on this feature and can be seen here **Experiment *H_2***. This test was aimed at finding the search terms in the recommended APIs to verify that the recommended APIs actually contain the terms the user is looking for. The recommended API does indeed contain the search words but the algorithm is very stiff and could need tuning to capture the essence of what the user truly is after. The lack of type comparison also makes the results somewhat ambiguous.

7.2 User Service

So how well does the service serve the requirements of the end user? First and foremost the input data gathered are used in a way that does not really distinguish documentation text from queries. The result of this is that the score created for the input is not representative for the required input arguments that the user might be able to apply. What the input score actually might be able to represent is the nature of the APIs it recommends but the URI query is still not known to the end user until manually investigated. The positive side is that example queries get dissected so smaller chunks of potential useful query keys get included in the representation.

The output assessment is fairer but the missed opportunity of coupling the key with intended value typing lessens the accuracy and overall use. Additionally, the source of the schema should be presented alongside the source of the API documentation page if found.

What the representation of both input and output sorely lack is some kind of explanation on how the score gets created through some kind of anchor text. The anchor text should also give a brief overview of what the API does.

The positive side of the argument is that the service actually reduces the number of documentation pages for the user and presents them in an orderly fashion. So it can be used to reduce the development delay for developers that seek open APIs if the discussed problems get addressed.

7.3 Future Work

There are features left out of the design and implementation that should help to serve the service in a positive light.

One such feature would be to have a mechanism that reconstructs URI queries fit for the given APIs. Much of the required information for this feature is present in the HTML code through URI examples and through the question mark separator with the ampersand and semicolon delimiters. The reconstructed URIs could be tested with the Schema coupled with the API to verify that the return form is of use to the end user.

Another feature could take advantage of the previous suggestion. This feature would include a translation table that converts API specific queries into standardized vectors. These vectors should be applicable both ways so a single query potentially can reach out to a series of APIs with different queries with the same meaning. An example of this would be two URIs with a single timestamp query key. The first query could be given an arbitrary timestamp of "1994-11-05T08:15:30-05:00" that corresponds to November 5, 1994, 8:15:30 am, US Eastern Standard Time. This timestamp could be translated to a standardized form that the mechanism translates to other queries acceptable for other URIs such as query 2 that accept "1994-11-05T13:15:30Z" that corresponds to the same instant. The use of this for the end user is that the mechanism could let the user run a single query search that yield a series of possible responses that can be listed as possible outcomes. The problem with this feature is that it requires a lot of manual work in order to get it going without delving heavily into latent semantic analysis or machine-learning.

There are currently several similar variations of the exact same word present in the Key Phrase Database such as "latitude" and "latitude,". A table of aliases should be created that convert entries like these into a single entry where redirection is used to store all information in a centralized location.

8 Conclusion

This thesis has had a focus on a design aimed at simplifying the development process for developers in search of open APIs through reducing the span of API documentation exploration as the **Problem Statement 1.1** declares: *Design a set of automatic overview mechanics presenting resource location and relevance on key phrases related to open APIs as a mean to reduce development delay due to open API documentation exploration.* In order to reduce their development delay, the design consist of two main sections; the Crawler and the Data Handler. Where the Crawler is a purpose-built web crawler that gathers API documentation and connected schema. While the Data Handler treats the gathered data which gets utilized together with input from an end user to create a set of API recommendations for that given user.

The findings of this thesis reveal that the structure of the design, in fact, can be used to create a service that helps developers in search of APIs. As the resulting API recommendations that get created at the end of the computational chain have been verified to conform to a set of search keys. The service created a demonstration of this claim is however not without flaws. An increase in crawled domain territories increases the crawl time in tandem with the increased reach of the service, thus causing a scalability problem. Mono threading of the service is also a culprit for the prolonged execution time. Additionally, the separation between any deployer and end user are not currently factual as every component of the service is currently centralized.

Considering the aspects mentioned above and by keeping in mind that the implementation is a proof of concept have the thesis provided one possible solution to reducing manual API documentation exploration time. And as a whole does the solution show promise to become a viable option through some addressment of the outlined problems.

Bibliography

[1] ProgrammableWeb, "Chart of web api growth from 2005 through 2013 (source: Programmableweb.com)." `https://www.slideshare.net/programmableweb/web-api-growthsince2005?ref=https://www.programmableweb.com/api-research`, Apr 2014. Accessed: 2017-16-11.

[2] W. Santos, "Programmableweb api directory eclipses 17,000 as api economy continues surge (source: Programmableweb.com)." `https://www.programmableweb.com/news/programmableweb-api-directory-eclipses-17000-api-economy-continues-surge/research/2017/03/13`, Mar 2017. Accessed: 2017-16-11.

[3] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do api changes trigger stack overflow discussions? a study on the android sdk," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, (New York, NY, USA), pp. 83–94, ACM, 2014. Accessed: 2017-11-15.

[4] M. Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.

[5] T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann, "World-wide web: The information universe," *Internet Research*, vol. 20, no. 4, pp. 461–471, 1992.

[6] R. T. Fielding and G. Kaiser, "The apache http server project," *IEEE Internet Computing*, vol. 1, no. 4, pp. 88–90, 1997.

[7] V. Anand, "Creating a rest service using asp.net web api." `http://prideparrot.com/blog/archive/2012/3/creating_a_rest_service_using_asp_net_web_api`, Mar 2012. Accessed: 2017-11-15.

[8] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000.

[9] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology*, vol. 2, pp. 115–150, May 2002.

[10] S. Spetka, "The tkwww robot: beyond browsing," in *Proceedings of the 2nd. WWW conference*, vol. 94, 1994.

[11] S. M. Mirtaheri, M. E. Dinçtürk, S. Hooshmand, G. V. Bochmann, G.-V. Jourdan, and I. V. Onut, "A brief history of web crawlers," in *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 40–54, IBM Corp., 2013.

[12] T. Yoke Chun, "World wide web robots: an overview," *Online and CD-Rom Review*, vol. 23, no. 3, pp. 135–142, 1999.

[13] A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts*. New York, NY, USA: McGraw-Hill, Inc., 6 ed., 2011.

[14] P. A. Ly, C. Pedrinaci, and J. Domingue, "Automated information extraction from web apis documentation," in *International Conference on Web Information Systems Engineering*, pp. 497–511, Springer, 2012.

[15] T. Xie and J. Pei, "Mapo: Mining api usages from open source repositories," in *Proceedings of the 2006 international workshop on Mining software repositories*, pp. 54–57, ACM, 2006.

[16] C. Parnin and C. Treude, "Measuring api documentation on the web," in *Proceedings of the 2nd international workshop on Web 2.0 for software engineering*, pp. 25–30, ACM, 2011.

[17] D. Crocker, "Standard for the format of arpa internet text messages," 1982.

[18] D. Connolly, B. Bos, Y. Koike, and M. Holstege, "A conversion tool from dtd to xml schema," Apr 2000.

[19] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley, "Network performance effects of http/1.1, css1, and png," in *ACM SIGCOMM Computer Communication Review*, vol. 27, pp. 155–166, ACM, 1997.

[20] D. Gourley and B. Totty, *HTTP: the definitive guide*. " O'Reilly Media, Inc.", 2002.

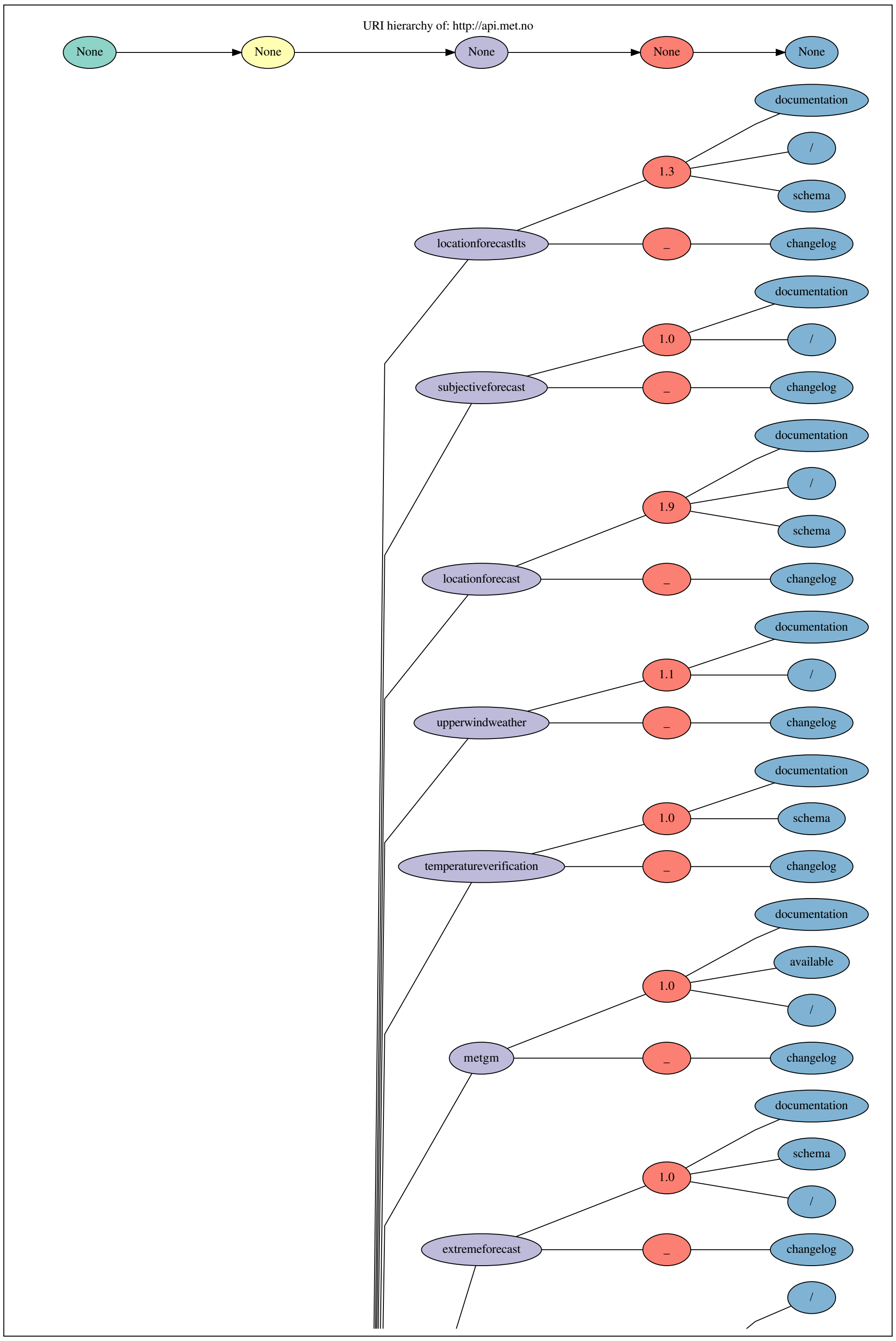
Appendices

Appendix B.1 Contain results from the key phrase lookup: Key phrases used in calculation of the Documentation Score: "wind", "latitude" and "latitudes". Key phrases used in calculation of the XML Schema Score: "latitude" and "wind".

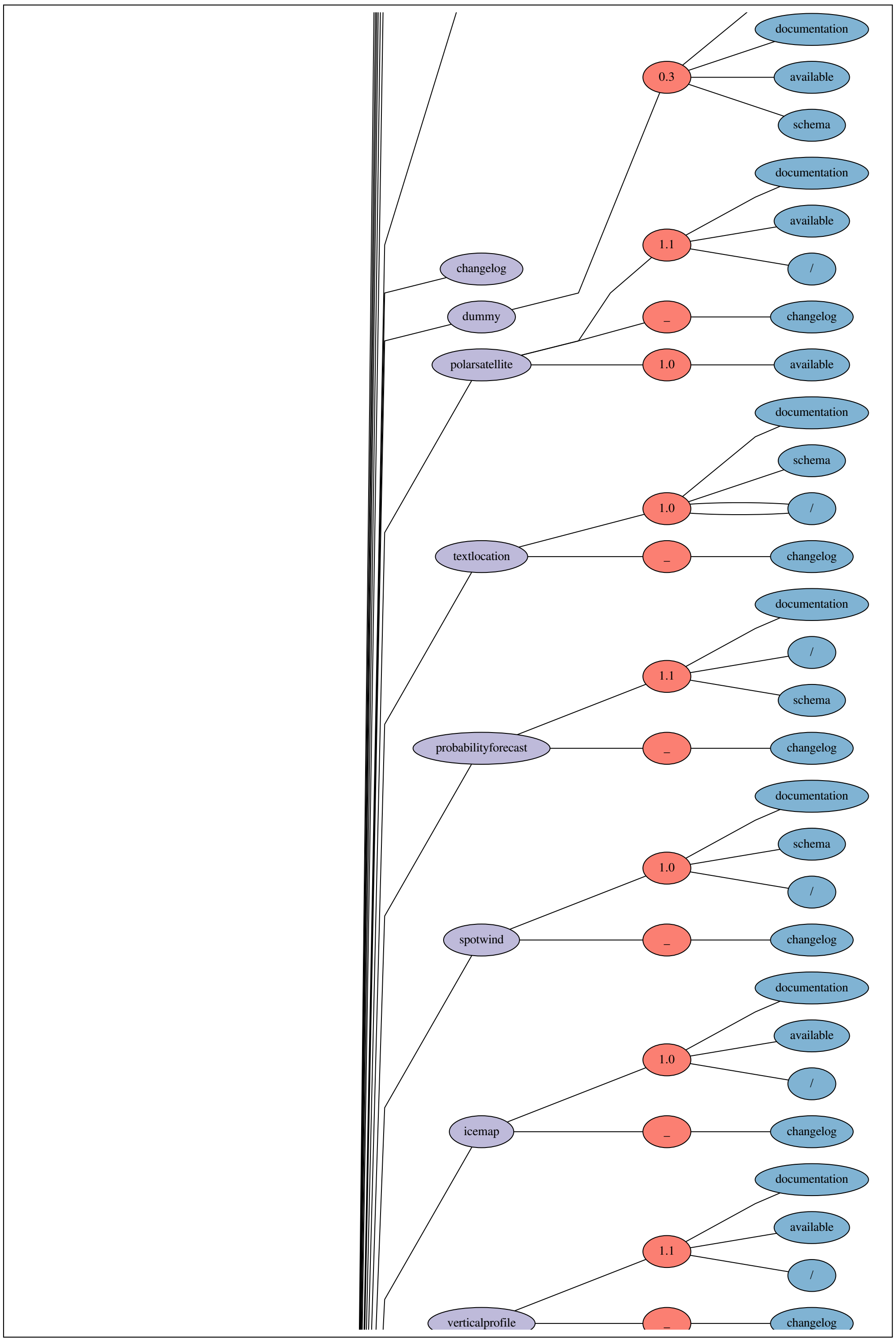
Appendix C.2 and **Appendix C.5** contain outliers from these sources:

1. http://developers.gettyimages.com/api/docs/v3/downloads/videos/id/post
2. http://developers.gettyimages.com/api/docs/v3/downloads/images/id/post
3. http://developers.gettyimages.com/api/docs/v3/downloadworkflow.html
4. http://developers.gettyimages.com/api/docs/v3/oauth2.html

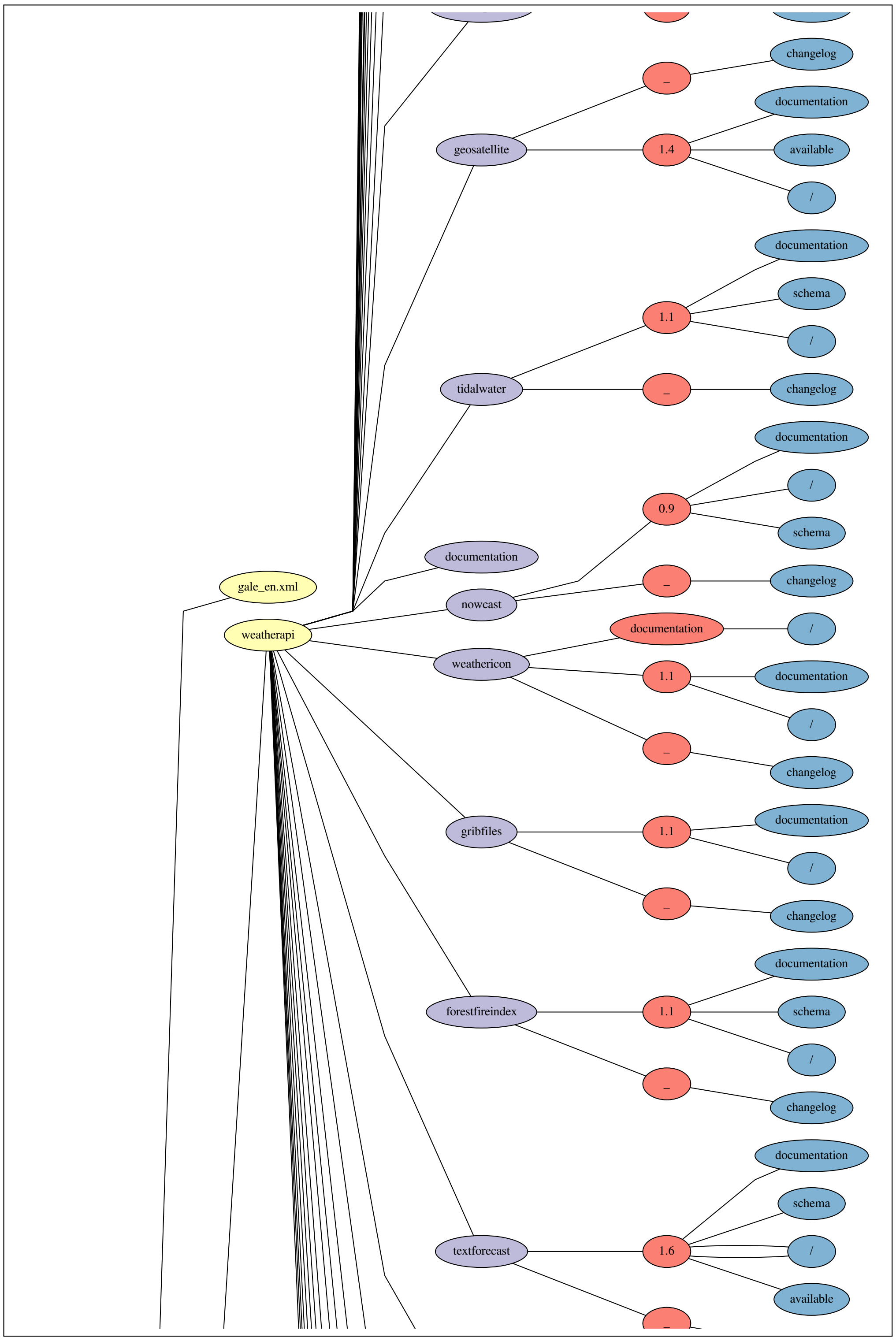
A Results: Data Acquisition



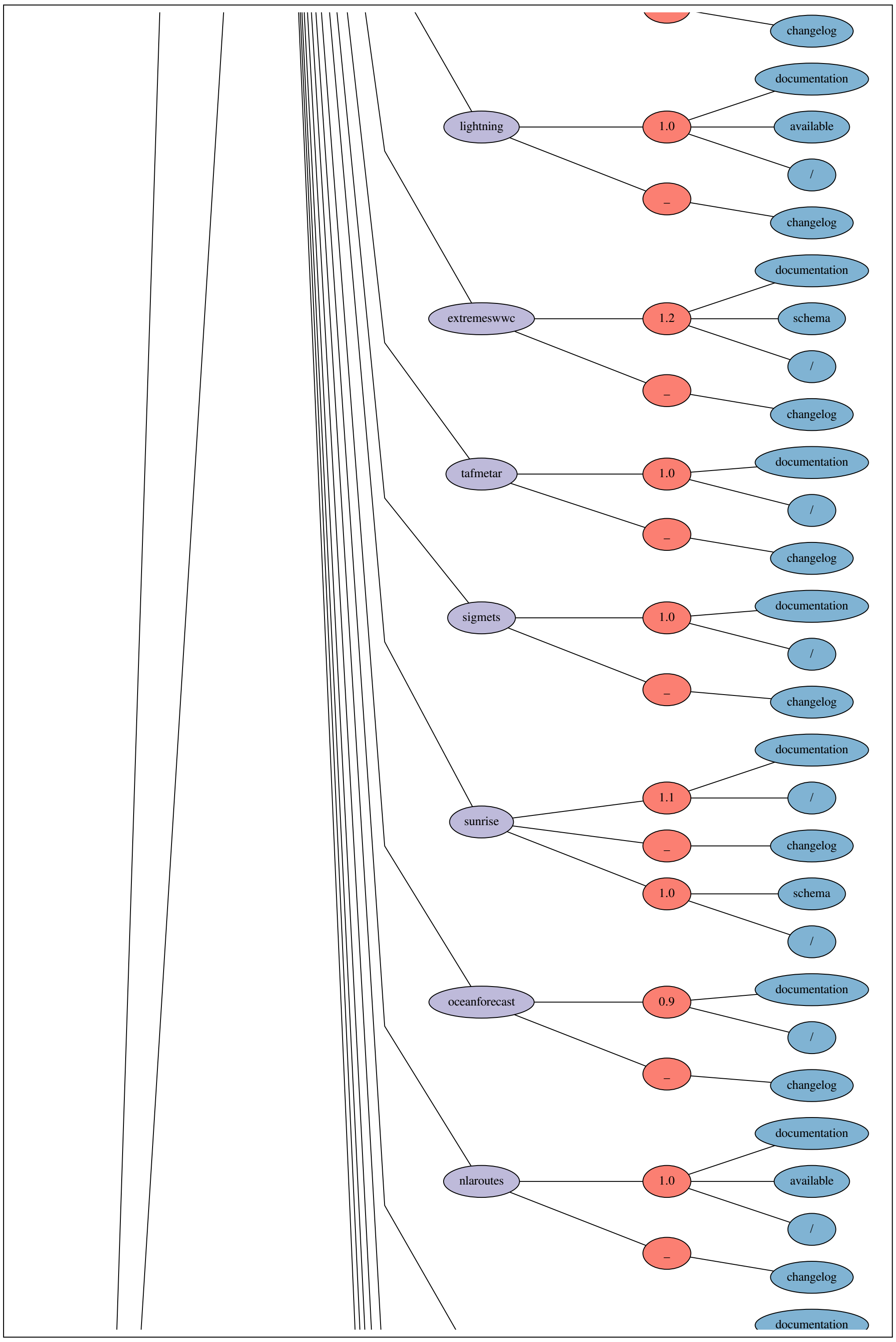
(a) Subfigure of Figure A.1



(b) Subfigure of Figure A.1



(c) Subfigure of Figure A.1



(d) Subfigure of Figure A.1

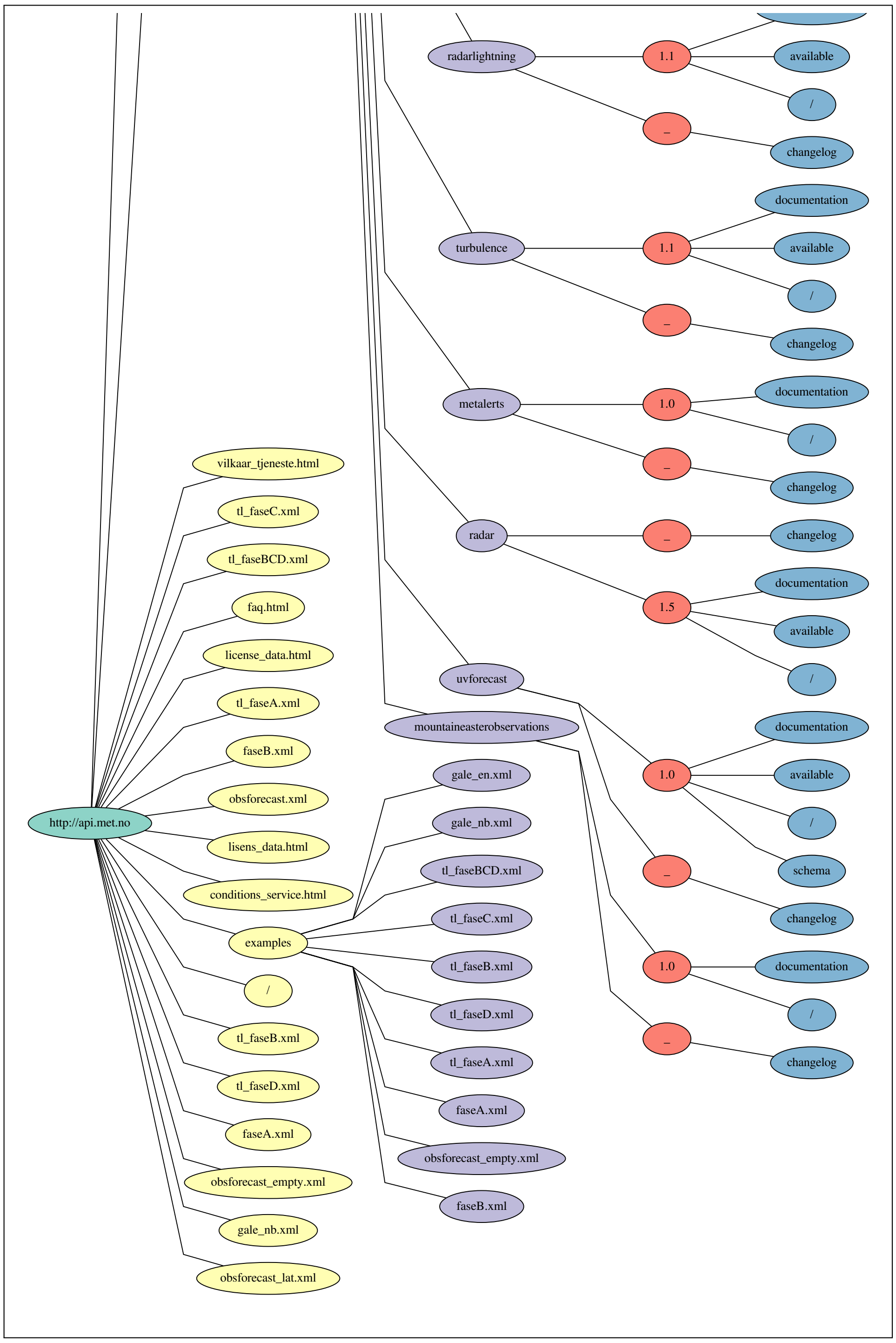


Figure A.1: URI Hierarchy representaion of the api.met.no domain.

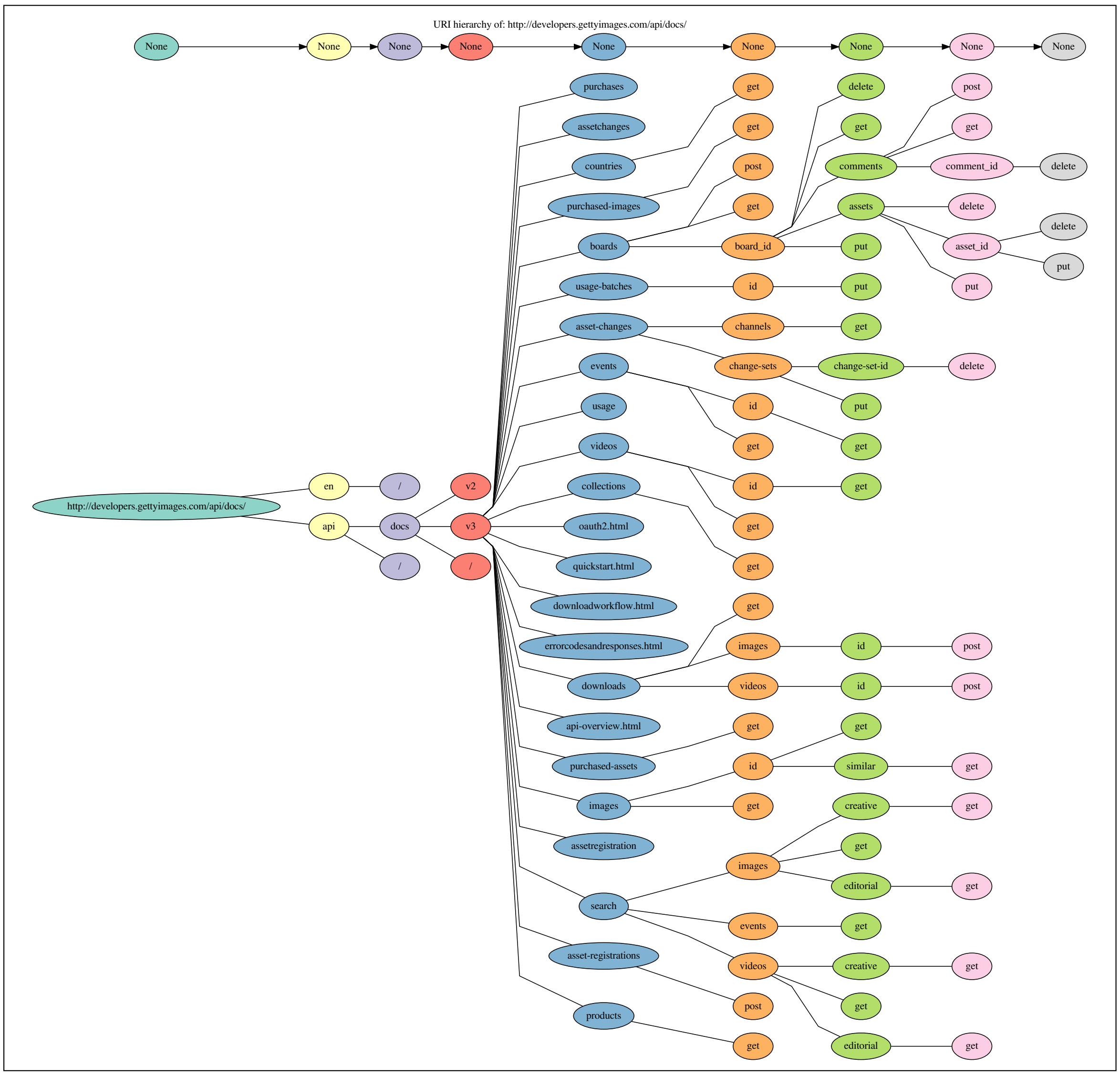


Figure A.2: URI Hierarchy representaion of the developers.gettyimages.com domain.

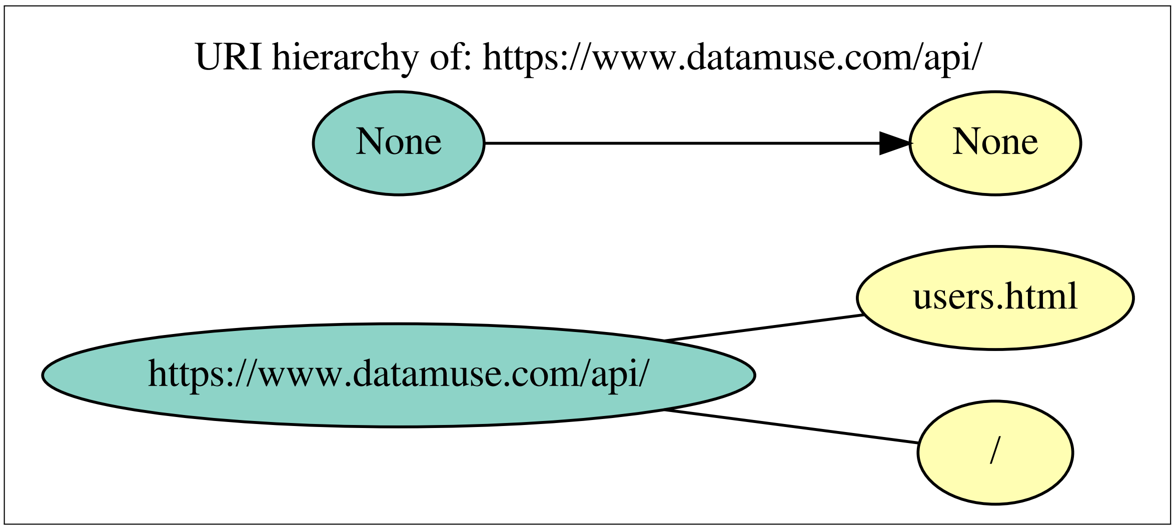

URI hierarchy of: https://www.datamuse.com/api/
None
None
users.html
https://www.datamuse.com/api/
/

Figure A.3: URI Hierarchy represenstaion of the crawling www.datamuse.com domain.

B Results: Data Handling

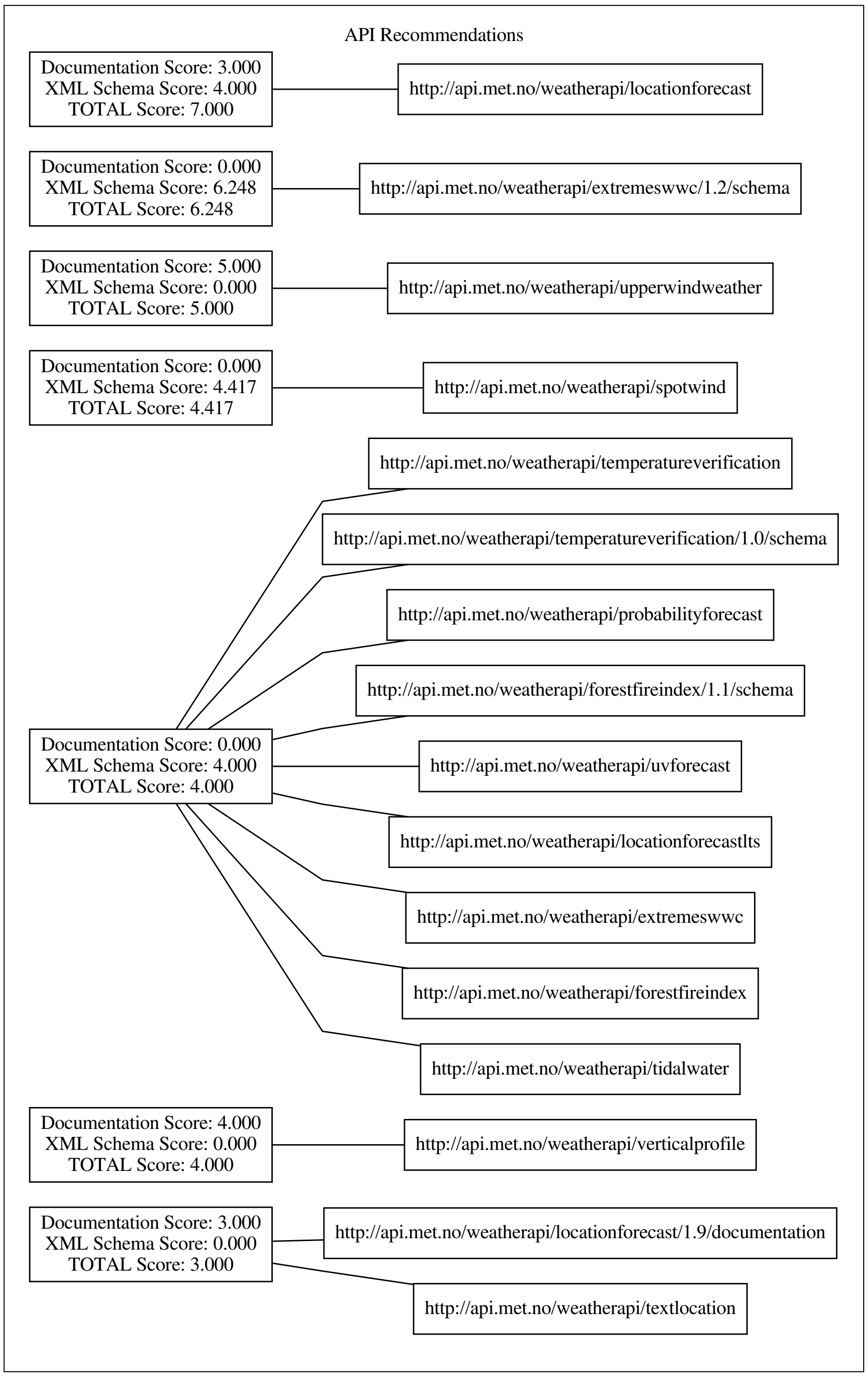

API Recommendations
Documentation Score: 3.000
XML Schema Score: 4.000
TOTAL Score: 7.000
http://api.met.no/weatherapi/locationforecast
Documentation Score: 0.000
XML Schema Score: 6.248
TOTAL Score: 6.248
http://api.met.no/weatherapi/extremeswwc/1.2/schema
Documentation Score: 5.000
XML Schema Score: 0.000
TOTAL Score: 5.000
http://api.met.no/weatherapi/upperwindweather
Documentation Score: 0.000
XML Schema Score: 4.417
TOTAL Score: 4.417
http://api.met.no/weatherapi/spotwind
http://api.met.no/weatherapi/temperatureverification
http://api.met.no/weatherapi/temperatureverification/1.0/schema
http://api.met.no/weatherapi/probabilityforecast
http://api.met.no/weatherapi/forestfireindex/1.1/schema
Documentation Score: 0.000
XML Schema Score: 4.000
TOTAL Score: 4.000
http://api.met.no/weatherapi/uvforecast
http://api.met.no/weatherapi/locationforecastlts
http://api.met.no/weatherapi/extremeswwc
http://api.met.no/weatherapi/forestfireindex
http://api.met.no/weatherapi/tidalwater
Documentation Score: 4.000
XML Schema Score: 0.000
TOTAL Score: 4.000
http://api.met.no/weatherapi/verticalprofile
Documentation Score: 3.000
XML Schema Score: 0.000
TOTAL Score: 3.000
http://api.met.no/weatherapi/locationforecast/1.9/documentation
http://api.met.no/weatherapi/textlocation

Figure B.1: Example results from the key phrase lookup.

C Misc

Installations of modules used for creation and use of the crawler and handler.

1. Install Ubuntu 16.04 or any appropriate substitution.
2. Python 3.5 or higher (bundled with Ubuntu, Required).
3. Graphviz installation(Required) e.g.:
   pip install graphviz
   pip install pygraphviz
4. Beautiful Soup installation(Required) e.g:
   pip install beautifulsoup4
5. Natural Language Toolkit installation(Required) e.g:
   sudo pip install -U nltk
6. JSON Schema installation e.g:
   pip install jsonschema
7. Tidy HTML tool installation e.g:
   sudo apt install tidy
8. Memory Profiler installation e.g:
   pip install -U memory_profiler
   python setup.py install
9. Psutil Process utility installation e.g:
   sudo pip install psutil
10. NumPy matplotlib e.g:
    sudo pip install -U matplotlib
11. NumPy installation e.g:
    sudo pip install -U numpy
12. UJSON installation e.g:
    pip install ujson
13. Lxml installation e.g:
    pip install lxml

Figure C.1: Setup Installations

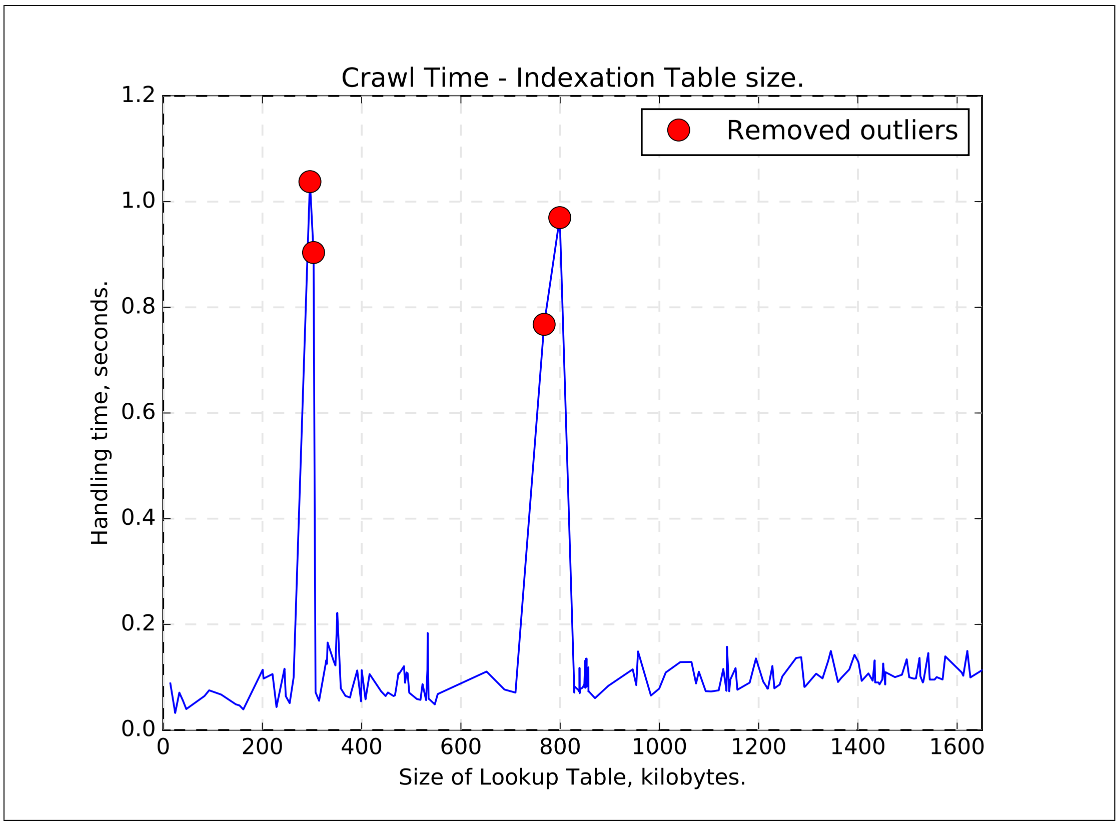


Figure C.2: Outliers - Crawl Time, Indexation Table size.

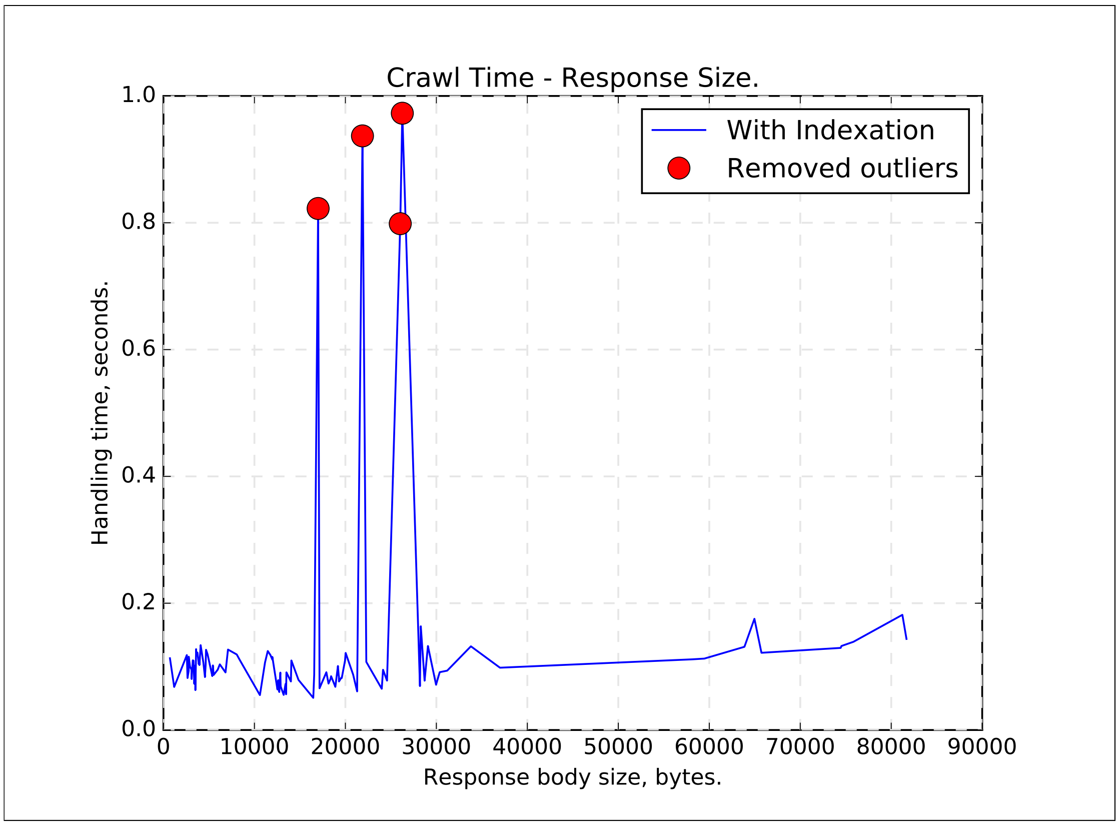


Figure C.3: Outliers - Crawl Time, response body size.

```json
{
  "$schema": "http://json-schema.org/draft-06/schema#",
  "description":"Outer list takes indexed words, Middle
   list APIs and Inner list URLs and occurance",
  "patternProperties": {
    "[-~]*$": {
      "type": "object",
      "patternProperties": {
        "[-~]*$": {
          "type": "object",
          "patternProperties": {
            "URL": {
              "type": "string"
            },
            "occurance": {
              "type": "number"
            }
          },
          "additionalProperties": false
        }
      },
      "additionalProperties": false
    }
  },
  "additionalProperties": false
}
```

Figure C.4: JSON Schema for Indexation Table validation.

```
{
  "$schema": "http://json-schema.org/draft-06/schema#",
  "description": "Outer list contain URIs for API and
   Inner list show indexed words from XML Schema",
  "type": "object",
  "patternProperties": {
    "[-~]*$": {
      "patternProperties": {
      }
    }
  },
  "additionalProperties": false
}
```

Figure C.5: JSON Schema for XML Schema Table validation.