WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

Bachelor Thesis

Open Dataset Archive

Scalable dataset crawling with efficient archiving and the investigation of changes between versions.[1]

Thomas Weber

Date of Birth: 05.03.1996
Student ID: 01553755

Subject Area: Information Business

Studienkennzahl: 033 561

Supervisor: Dr. Neumaier Sebastian

Co-Supervisor: Univ.Prof. Dr. Polleres Axel

Date of Submission: September, 20th 2020

Department of Information Systems and Operations, Vienna University of Economics and Business, Welthandelsplatz 1, 1020 Vienna, Austria

[1]The results of this bachelor thesis have been published as a resource track paper[1] in ISWC2020.

DEPARTMENT FÜR INFORMATIONSVERARBEITUNG UND PROZESSMANAGEMENT DEPARTMENT OF INFORMATION SYSTEMS AND OPERATIONS

Contents

List of Figures

List of Tables

Listings

Abstract

In this paper we present the Open Dataset Archive (ODArchive), a dataset crawling and archiving infrastructure which regularly crawls and indexes Open Data (OD) resources. It performs basic data cleansing on known formats, and provides unified access to a large corpus of structured data from OD portals through APIs that allow flexible filtering, e.g. through SPARQL queries over the meta-data, for on-the-fly generation of specific sub-corpora for repeatable experiments.

The ODArchive enables us to investigate the changes and development of web resources over time and gather statistics, metrics and insights of open datasets available on the Internet. Our work describes the requirements and the technologies used to implement the ODArchive.

It is available at: `https://archiver.ai.wu.ac.at/`.

1 Introduction

The initial vision of the World Wide Web in 1989 by Tim Berners-Lee was a system of interlinked documents, to solve the problem of locating information on distant machines. To do so, he developed the Hypertext Markup Language (HTML) which describes the structure of a document and allows to add hyperlinks to other documents. Over the years the Web became the main tool to share and spread information and has seen an enormous growth. However, the Web also became broader in terms of available content: while the main part of the Web consists of HTML documents, readable by humans, we can also see a trend towards publishing datasets openly available on the Web and towards a "Web of Data"[2]. More and more datasets get published on the Web, for instance as "Open Data" – freely available to everyone to use and re-publish without restrictions – which is typically available on governmental data portals, or via community portals, such as the data science platform "Kaggle" where community members can share datasets with each other.

As the Web became the main tool for publishing and sharing information worldwide, there also emerged projects for storing and archiving all this published information; most important the Internet Archive[3] and the European Web Archive: while the latter is an archive only for the EU institutions, agencies and bodies, the former is an approach to crawl and archive all content on the web (see Section 2.3 for more details). The motivation behind these archiving approaches is to provide a digital library of all documents available online, as for instance the state archive or public libraries of countries would do for all print/physical media. These digital archiving solutions however, only access websites, i.e. the HTML resources and do not archive other resources, such as CSV tables, XML, JSON, or RDF files, or even PDF documents. The growing amount of such data freely accessible on the web and the fact that this data is changing over time aroused our interest in the metrics of web resource changes and large scale archiving of *Open Data*.

The Open Data (OD) movement, mainly driven by public administrations in the form of Open Government Data has over the last years created a rich source of structured data published on the Web, in various formats, covering different domains and typically available under liberal licences. Such OD is typically being published in a decentralized fashion, directly by (governmental) publishing organizations, with data portals, often operated on a national level as central entry points. While these portals provide somewhat standardized metadata descriptions and (typically rudimentary, i.e. restricted to

[2] https://www.w3.org/2013/data/

[3] https://archive.org/

metadata only) search functionality, the data resources themselves are available for download on separate locations, as files on specific external download URLs or through web-APIs, again accessible trough a separate URL.

Despite there is initial work on harvesting, integrating and *monitoring meta-data from over 260 OD portals* for several years now in the Portal Watch project [2], we want to provide unified access to this rich data source. Underlining the increasing importance of providing unified access to structured data on the Web, Google recently started a dataset search [3] facility, which likewise indexes and unifies portal meta-data adhering to the Schema.org [4] vocabulary in order to make such meta-data searchable at Web scale. This OD *meta-data* is well investigated in terms of searchability [3] or quality, [2] but the underlying referenced *datasets*, i.e. the actual resources themselves, and their characteristics are still not well understood:

- What kinds of data are published as OD?
- How do the datasets themselves develop over time?
- How do the characteristics of datasets vary between portals?

In order to enable answering such questions, our goal in the present thesis is to provide a resource in terms of a dynamically updated corpus of datasets from OD portals, with unified access and filtering capabilities, that shall allow both profiling and scientific analyses of these datasets. To this end we have created, on top of the Portal Watch framework, a dataset crawler and archiver which regularly crawls and indexes OD resources, and provides unified access to a large corpus of archived data through APIs that allow flexible filtering, e.g. through SPARQL queries over the meta-data, for on-the-fly generation of specific sub-corpora for experiments. We deem this project particularly useful as a resource for experiments on real-world structured data: to name an example, while large corpora of tabular data from Web tables have been made available via CommonCrawl [5], the same is not true for tabular data from OD Portals.

We fill this gap by presenting the ODArchive, an infrastructure to crawl, index, and serve a large corpus of regularly crawled structured data from (at the moment) 137 active portals. We describe the challenges that needed to be overcome to build such an infrastructure, including for instance automated change frequency detection in datasets, and make the resource available via various APIs. Specifically, we make the following concrete contributions:

- We present a detailed architecture of a distributed and scalable Dataset Archiver. The archiver is deployed at `https://archiver.ai.wu.ac.at`, and the software is openly available on GitHub[4] under the MIT license.

[4] `https://github.com/websi96/datasetarchiver`

- Using the introduced archiver, we regularly collect and archive – based on an approximation of the change rates – a large corpus of datasets from OD sources, and make the whole corpus, including the archived versions available via different APIs, incl. download access to subsets of the corpus configurable via SPARQL queries.

Our main goal is to archive any kind of data and to investigate the changes and the overall development of single resources over time. We also designed the OD Archive to be easily scalable and distributable over different clusters to handle the growing amount of data. In our work we also describe the implementation of the crawler and the challenges like file compression, change analysis, crawling frequencies or scalability while crawling different data types of different sizes from different resources.

1.1 Problem Overview

Despite there are solutions for versioning and storing data (Git, Apache Subversion), handling large file sizes and the vast amount of data on the web cannot be covered by traditional solutions. To implement such a crawler and to archive different versions of large files, we had to cover the problems of data type detection, change detection, workload-management, scalability and the storage-recreation trade-off further explained in this section.

In order to overcome all this problems, we present our solutions and implementation in Section 4. In Section 4.3 we provide our solutions for workload-management and scalability. Section 4.4 gives an insight of how we handle the crawled data.

1.1.1 Data Type Detection

Because some providers do not attach any information about the type of the sent file in the HTTP headers, the detection of the correct file type is hard to implement. It might also be the case, that the received file is compressed, which enforces our crawler to decompress the file for further analysis of the filetype. It is also not guaranteed that file endings lead to correct file types, as well as there might not even be file endings or names.

The data could also be unstructured, in which case the analysis of the data would require extensive parsing approaches to get to know the crawled data. To better understand structured and unstructured data, we will give a short overview about different data types in Section 2.1.

1.1.2 Detection of Changes

Defining an efficient way to recognize changes is a key problem while minimizing processing power on handling this great amount of large files, because first and foremost the change frequency of an unknown resource must be calculated. In a second step a program must be able to recognize a change of a file and at last it must identify the changes themselves in detail.

The definition of the change frequency can either be fixed or dynamic, but there is no algorithm, which defines an accurate way of identifying the change-rate of resources on the web.[6]

1.1.3 The Storage-Recreation Trade-off

Another problem is the trade-off between the "Total Storage Cost" and the "Recreation Cost". Minimizing both is preferred, but either we use more storage and are able to recreate different versions of the data in a fast way or we use less storage and therefore it is slower to retrieve the data again. As Amol Deshpande [7] states, most of the variants of the problem of balancing these two costs are NP-Hard. This means that these problems may possibly not be verifiable in polynomial time and efficient heuristics are needed for approximation solutions to this problem.[8]

Also an appropriate compression or de-duplication method needs to be found in order to minimize the needed storage. Although this would minimize the storage cost, we do not focus on compression methods and de-duplication algorithms in our work. See how we handle our data in Section 4.4.

1.1.4 Workload-management and Scalability

The vast amount of data and their enormous file size challenge traditional software archetypes and the power and storage capabilities of single servers. Also the computational effort to calculate statistical output is a huge point of failure. Therefore the calculation of the correct crawling attempt frequency is needed to balance the workload.[6]

1.2 Research Question

Because of the various problems stated above the following research question is guiding our research:

- How can a system archive and version structured and unstructured data in a scalable and efficient way?

- To address this question we consider current archiving and versioning methods and how these systems detect the changes of files.
- We also analyze how a system can efficiently store and retrieve different versions of large data and how to manage the workload while crawling huge amounts of large files.
- Further we attempt to find metrics which can be gathered while downloading and archiving different file types.
- To make the archived datasets searchable and accessable we provide meta-information regarding the specifics of the archived versions. Using this auto-generated metadata the archive can directly be queried, accessed.

1.3 Thesis Structure

The remainder of this thesis is structured as follows: The analysis of background literature and state of the art solutions is presented in Section 2, the definitions of requirements and services in Section 3 and a detailed implementation documentation in Section 4. We present our findings while crawling about 1.2 Million datasets in Section 5 and a conclusion of our archiving service, which aims to tackle this question and provides an efficient and scalable way of crawling large amounts of data in Section 6.

The developed ODArchive can be accessed at `https://archiver.ai.wu.ac.at/`.

2 Preliminaries & Background Literature

This section gives an overview of the literature we reviewed and describes the state of the art concepts and solutions which need to be understood to follow along the implementation of the system.

2.1 Data Types

Investigating the Internet Archive[5] a bit further, they state to archive web pages, books, texts, audio and video files, images and software programs. In our case however we want to focus on textual and binary open datasets like CSV, JSON, XML, RDF or ZIP. According to Sint et al. [9] all data types can be split up into following three categories:

[5] `https://archive.org/about/`

2.1.1 Unstructured Data

Unstructured data does not have any kind of data schema and therefore has no identifiable structure. It is often referred as binary data. Examples for unstructured data are PDFs, images, videos and zip archives. Also web page content is a kind of unstructured data, of which the "waybackmachine" from the Internet Archive already provides versions based on snapshots.

2.1.2 Semi-structured Data

Semi-structured data on the other hand can be seen as some kind of tree-like data or as an unstructured table. Tree-like data can have the XML or JSON format and tabular data is mostly stored as Comma Separated Values (CSV).

All this semi-structured file types are basically just ordinary text files representing the data in a predefined way and currently there is no service providing access to versions of such file types. This leads to the fact, that semi-structured data is our main focus while crawling different resources.

2.1.3 Structured Data

Structured data has a predefined schema, which is mostly defined in a relational way or as graph data. Relational data can mostly be found in relational database systems and is often used in combination with object oriented programming.[9] Graph data can be represented with the Resource Description Framework (RDF) and is mostly referred to as linked data.

2.2 Architectural Hurdles

While investigating the topic of large scale data crawling, we came across difficulties we had to consider. Not only we must ensure politeness to several hosts, we also have to implement a metric to determine the change-rate of resources on the web. To be further able to enlarge our corpus we also have to take the scalability of our architecture into account.

2.2.1 Host Politeness

Unlike a normal user, our web crawler will access a web-host multiple times in parallel. This may lead to a denial of service, if it cannot handle enough requests at the same time. To not getting blocked by various hosts, the crawler must enforce politeness rules for each of the indexed datasets.

As Riemer [10] states, there are several ways not to overstrain a host while crawling, which they implemented in their approach. The best way is

to investigate and enforce the Robots Exclusion Protocol (robots.txt) rules, which define restrictions for web crawlers and the definition of individual crawling delays, to ensure politeness for each host.

Besides there are rules for crawlers, which are blocking them to access some specific sites. Google recently announced that the Noindex, Nofollow and Crawl-delay rules of the robots.txt are deprecated and therefor not used anymore by the Googlebot.[11]

2.2.2 Dynamic Crawl-Rate

To investigate changes over time, the data we want to analyze has to be crawled in a predefined interval. To determine this interval, Neumaier and Umbrich [12] propose three strategies.

Either the changes are propagated by the data provider in a push based way *(push-based change history)*, an information about the latest change time of a resource is available from the data source *(age sampling)*, or the newly downloaded data is compared with the older version of the same data *(comparison sampling)*.[12]

In our case only the comparison sampling makes sense, because not many people will propagate their data changes to the crawler, nor do some of them have the possibility to do this in an automated way. Also the latest change information in the HTTP Header lacks a broader acceptance on the web.

The frequency of crawling attempts influences the needed resources and bandwidth of the crawler. Distributing this frequency over the number of hosts a crawler has to crawl is therefore mandatory for the scalability of such a system.

2.2.3 Scalability

Speaking of scalability, one must know that a system is limited by resources in terms of hardware. Traditional software is executed on just a single server or computer and is therefore limited by the hardware of this single unit and cannot be easily extended.

For our crawler to be able to expand the number of datasets it is capable to crawl simultaneously and store on disk, the software architecture must provide a way to add hardware and increase the speed and capacity of the system preferable in a linear way. In the best case the resources scale themselves depending on the workload the crawler has to handle.

In Section 4.3.1 we state how we utilize kubernetes, containerization and mongodb to dynamically scale our infrastructure.

2.3 Related works on data archiving and versioning

As mentioned in the introduction, there already exist technologies, which crawl the web at large scale or provide solutions for versioning specific types of data. This section tries to give an overview of existing platforms, frameworks and technologies to crawl and interact with large data corpora.

2.3.1 Online Platforms

While searching for existing platforms, which have the same characteristics as our proposed archiving and versioning tool, we came across following four services:

Archive.org - WayBackMachine The Internet Archive is a non-profit organization to build a digital library of Internet sites. They state, that their mission is to provide universal access to all knowledge and began to archive the Internet in 1996. Therefore their Archive-It program identifies important web pages and stores them in a Wayback Machine.[6] Today their archive not only contains websites, but also books, texts, media content and software. It is one of the top 300 web sites in the world and their library occupies 45+ Petabytes of server space for a single copy of the data shown below:

- 330 billion web pages
- 20 million books and texts
- 4.5 million audio recordings
- 4 million videos
- 3 million images
- 200,000 software programs

In contrast to our OD approach, the Internet Archive pays special attention to books, web and television content. In 2009 they began to make selected U.S. television news broadcasts searchable by captions. This service allows researchers and the public to use television as a citable and sharable reference.[13]

Common Crawl The Common Crawl Foundation is a also a non-profit organization which tries to democratize the access to information on the web. Their vision is to make the internet a truly open web that is universally accessible, analyzable and allows open access to information for everyone. Common Crawl makes wholesale extraction, transformation and analysis of web data cheap and easy and their corpus contains petabytes of raw web page data, metadata extracts and text extracts. Although access to data

[6] https://archive.org/web/

they host on Amazon AWS is free, Amazons cloud platform can charge you while running analysis jobs directly against their data.[14]

Based on the Common Crawl, Lehmberg et al. [5] presented a large corpus of (typically much smaller) Web tables, consisting of 233 million content tables. They classified these tables as either relational, entity, or matrix tables depending on the orientation and structure of a table, detecting subheader rows/multi-tables and subject columns in a dataset. In future work, we want to apply this classification to our corpus – particularly to the tabular resources – in order to highlight and compare the differences of a corpus of Web/HTML and OD CSV tables. A survey on profiling relational data can be found in [15].

Kaggle Kaggle is a subsidiary of Google and also one of the biggest data-science plattforms on the Internet. Not only they provide collections of uploaded datasets, but also allow users to have fully functional web-based Jupyter[7] compute environments to play with selected datasets.

It is a perfect playground for machine learning enthusiasts and data scientists who simply want to share their work. Kaggle is also known for its Machine learning competitions and educational programs.[16]

DBpedia - WayBackMachine To archive data of Wikipedia Fernández, Schneider, and Umbrich [17] created the DBpedia WayBackMachine which provides the wayback functionality for just DBpedia at any selected time based on the revisions of their Wikipedia article.

2.3.2 Git and SVN

Although there exist several versioning software like Git and SVN, all of them fail on versioning large files, as described by Amol Deshpande [7] and Bhattacherjee et al. [18], because they only use fairly simple algorithms to compute deltas for compression. Therefore they are mostly used as versioning tools for large code bases of small textual files, but are unhandy in handling binary data.

Furthermore Git-LFS, the large file storage extension of Git is only capable of handling files with a maximum file size of 2 GB. Git-LFS stores the whole files externally multiple times and does not do delta-compression on this kind of files, which means it is not suitable to efficiently archive large content.[19]

[7] `https://jupyter.org/`

2.4 Preliminaries and Technologies used in this thesis

2.4.1 Databases

To efficiently store data, database management systems are used to handle its storage and retrieval. Commonly speaking there are two types of such. On the one hand there are relational database management systems (RDBMS) and on the other hand there exist document stores. Document stores are often referred to as NoSQL databases.

We discuss the main differences of these two approaches using the database systems PostgreSQL (RDBMS) and MongoDB (NoSQL).

Relational PostgreSQL is an open source feature rich object-relational database system. It can be queried with the Structured Query Language (SQL) and has powerful add-ons like the PostGIS geospatial database extender.

Although Bhardwaj et al. [20] state that Oracle flashback and PostgreSQL are capable of doing a traversal search of versions going back and forth in time, it is hard to efficiently store large amounts of data. Despite PostgreSQL is able to store large files as "pg_largeobject", it does not provide a way to efficiently version them with its built in TimeTravel functionality, because they split up the metadata and the data in two separate tables.[21]

(In the newest versions of PostgreSQL the TimeTravel functionality is not existent anymore.)

NoSQL MongoDB on the other hand is a document database with high flexibility in data schemes and scalability. It stores data in flexible, JSON-like documents and can be queried with an intuitive query language. The querying is more or less the same for the shell and JavaScript.

```
1 db.users.insert({
2     "_id" : ObjectId("5db8db42f5323152269c1d87"),
3     "name" : "Tom",
4     "dob" : ISODate("1996-03-05T10:07:22.010Z"),
5 })
6
7 // returns the dob of all documents with name "Tom"
8 let docs = db.users.find({"name": "Tom"}, {"dob": 1})
```

Listing 1: MongoDB querying example

But also following a document versioning pattern with a NoSQL database like MongoDB has its limits. A blog entry on MongoDBs website states that following three factors must be considered before using a NoSQL database:

- The documents do not have too many versions.
- There are not too many documents to version.
- Queries are mostly performed on the latest version.

This indicates that also MongoDB is not the best choice for the needs of archiving data which changes in a frequent way, but might be useful to archive the data our crawler should handle.[22] Section 3.1 gives an overview about the data being crawled.

MongoDB also provides a convention for storing large files in a MongoDB database. This convention is called GridFS and stores the files in a chunked way with a fixed chunk size, which we might further use to de-duplicate the data.

2.4.2 Programming Languages and Concurrency

The underlying programming language we are using in our case is JavaScript, because of its non-blocking, event-driven nature. Compared to the traditional blocking programming styles of sequential programming used in Python or Java, JavaScript is more suitable for concurrent web traffic.

A program is non-blocking, when it does not have to wait for functions to have finished their execution. Code can therefore be called asynchronously after a function has been invoked.[23]

Node.js As asynchronous event-driven JavaScript runtime built on Chrome's V8 JavaScript engine, Node.js is perfectly suitable to develop scalable network applications.[24]

To access different open source dependencies, NPM provides packaged modules of code for Node.js. Furthermore TypeScript can be used to ensure type safety. TypeScript is a superset of JavaScript and uses a compiler to compile the source code to plain JavaScript. TypeScript makes intense use of types and definitions, which help making less mistakes while coding by providing instant errors from the compiler.[8]

I/O - Handling and Threading In our case the non-blocking programming style of JavaScript dramatically improves the time to download the data compared to blocking code like Python. When using non-blocking code, database access for example gets much faster, because each download invokes a database request and the further code can be executed in parallel to this

[8] `https://www.typescriptlang.org/`

requests. Here is a little example from the Node.js website to explain this a bit further:

> "As an example, let's consider a case where each request to a web server takes 50ms to complete and 45ms of that 50ms is database I/O that can be done asynchronously. Choosing non-blocking asynchronous operations frees up that 45ms per request to handle other requests. This is a significant difference in capacity just by choosing to use non-blocking methods instead of blocking methods."[23]

Threading or multithreading on the other hand is a way to parallelize processes on the operating system level and not on the runtime side as in Node.js. A Python program can be parallelized with multithreading. Threading is more or less limited by the CPU cores of the underlying operating system, because it utilizes processing power from each core.

Although Python is fast when multithreading is used, a Node.js program can nowadays also be spanned across multiple processor cores and is therefor easily scalable.[23]

2.4.3 Kubernetes, NGINX Ingress and Reverse Proxy

Kubernetes It is often referred to as 'k8s' and provides an open-source system to automatically deploy, scale, and manage containerized applications. It can manage multiple clusters and interlink them over the network interface.

NGINX Ingress It is a HTTP load balancer for applications, represented by one or more services on different nodes. The routing is controlled by rules defined on each Ingress resource.[25]

NGINX Reverse Proxy It is responsible for load-balancing HTTP requests and database connections from external IPs to the internal exposed Ingress services. The reverse proxy is therefore the main entrypoint for external connections and as such not only provides load balancing, but also functions as another layer of security.

3 Requirements and Services

Every good system starts with a plan. Therefore we decided to take a two step approach to accomplish our goals. At first we defined the primary

requirements and additionally we came up with secondary ones, to enhance the crawler even further.

3.1 Primary Requirements

Based on the previous examination of state of the art solutions and related work we defined requirements for our scope of data and sources, the scalability, the politeness to our hosts and our must have api endpoints.

Scope According to the study of Mitloehner et al. [26] 10 percent of CSV files from 232 OD portals have been labelled as CSV files and only 50 percent of them could have been parsed. Furthermore the HTTP response header of only 50 percent of all CSV files has been specified correctly, which indicates that its hard to identify the data type before downloading the whole file.

By just crawling CSV files we might lose data which could eventually be used, if it is cleaned and further analyzed by another system. Therefore we decided that the crawler must be able to handle different types of data. Any given data type must be supported. Just for example, the crawler must be able to handle PDF, JPEG, XLSX, ZIP and any textual-file like CSV, JSON, XML or RDF, to list just some of them.

In Section 5 we present the distribution of all the different filetypes our crawler downloaded by the time of writing this work.

Source To feed our crawler with resources, we rely on the work of Neumaier, Umbrich, and Polleres [2] and their OD Portal Watch. This collection of portals and their datasets provides us with a large corpus of urls to index in our crawler.

The crawler must therefore be able to regularly index urls from their sparql endpoint on `https://data.wu.ac.at/portalwatch/sparql`.

Scalabilty In order to create a system that scales to the huge amount of datasets and their size we have to divide our system into different logical subsystems. Umbrich, Mrzelj, and Polleres [6] propose to design such an infrastructure as some kind of P2P network. They also state, that it must be able to crawl all the resources in a parallel, dynamic and polite manner. It also must easily be extendable in terms of needed resources to increase for example its storage capacity.

Host Politeness In terms of concurrent request per host, our crawler must be polite to the hosts of the resources. To ensure our crawler does not get

blocked by certain hosts, a minimum of politeness is required to fetch all resources from the web. Therefore we decided that our crawler must only crawl one file from a host at a time, but must parallelize different hosts, until we can enforce enhanced politeness rules. (see Section 3.2)

Profiling To ensure quick access for basic statistics about the data corpus, a mechanism like pre-calculation or indexing of data profiles is needed. The user must be able to quickly find information about different profiles. We therefore try to implement some profiles defined by Mitloehner et al. [26] and Neumaier, Umbrich, and Polleres [2] e.g.:

- different types of data
- file size
- size distribution
- number of versions
- number of changes
- versatility of data

We also want to investigate tabular data and try to guess their column types. This will come in handy, when trying to find inter-referencing tables in our corpus. Read more about data analysis in Section 4.4.1.

3.1.1 Application Programming Interface

Another primary requirement is a Representational State Transfer Application Programming Interface (RESTful API). It is used to further develop applications on top of the underlying crawler architecture without the need to know the crawling software itself in detail.

To ensure compatability with other programs, our crawler needs a post endpoint to add resources, some get methods to retrieve files, a crawl endpoint to tell which resources must be crawled and some stats endpoints to have some basic knowledge about the data corpus.

The actual implementation reference of the API can be found in Section 4.2.

3.2 Secondary Requirements

In a second step we defined further requirements to have guidelines to keep the direction of our work in alignment with the goals we want to achieve with this project.

Arbitrary URLs Our crawler must be able to parse URLs from just any website and must be able to guide its way through the entangled connections of the interconnected web of data.

Data Quality Metrics The crawler must in a second step be able to combine his data resources with the data quality metrics from the work of Neumaier, Umbrich, and Polleres [2] about the "Automated Quality Assessment of Metadata across OD Portals".

Enhanced Host Politeness Enhanced politeness means that the crawler not only is able to parse the robots.txt, but further includes metrics to enrich the politeness and also the efficiency of the crawling process itself. This could be achieved by analyzing the content of each host and providing different heuristics taking the crawled content into account.

Web Interface Another feature of an upcoming version must be a web interface to provide an easy way for users to add and retrieve data to and from the crawler.
The previous mentioned API is a preparational step to accomplish this requirement.

4 Implementation

Based on the recommendations from Umbrich, Mrzelj, and Polleres [6] about capturing and preserving changes on the Web of Data, we decided to divide our infrastructure into three different layers:

1. *Network*: Kubernetes orchestrates our containerized software and NGINX balances the load on the cluster.
2. *Storage*: MongoDB stores all datasets as chunked binaries along with their associated metadata.
3. *Scheduling*: Our Client-Server architecture written in Node.js handles the crawling and scheduling components.

The client and the server are Node.js Express Applications which consist of an API and websocket capabilities. The clients connect to the scheduling master server component via a socket routed through NGINX Ingress.

Every crawling interval the scheduler asks the clients via a broadcast how many datasets they are able to crawl at the moment and responds with the requested amount of *dataset ids* each client is capable to handle. The client then downloads/crawls the requested resources from the web and store them in our sharded MongoDB.

Our software architecture follows the Model-View-Controller (MVC) pattern. We also implemented the controller functions as services to exclude the business logic, in our case the crawling logic, from the controlling logic. This makes testing our software easier.[27]

When a Client is requested to crawl a dataset, the functions in listing 2 are getting invoked. After the download of the file we compare the hashes of the old and the new version. We further calculate the next crawl and then release the dataset and host again. If an error occurs in this procedure, we add this error to the dataset metadata and calculate the next crawl as if the dataset has not changed. Then we again release the dataset and host.

```
class Crawler() {
  dataset = db.dataset.getCrawlable()
  async start() {
    try {
      await this.download()
      this.dataset.crawl_info.currentlyCrawled = false
      const hasChanged = await this.checkHash()
      this.calcNextCrawl(hasChanged)
      await this.dataset.save()
      await db.host.release(this.dataset.url.hostname)
      if (hasChanged) this.populate_sparql_endpoint()
      return true
    } catch (error) {
      this.dataset.crawl_info.currentlyCrawled = false
      this.addError(error)
      this.calcNextCrawl(false)
      await this.dataset.save()
      await db.host.release(this.dataset.url.hostname)
      return false
    }
  }
}
```

Listing 2: Main crawling functions

4.1 Architecture

The whole system runs on a cluster consisting of three nodes managed by Kubernetes. An Ingress service of NGINX handles the load balancing inside the cluster and an external NGINX reverse proxy in front of the cluster is responsible for the connections from outside the cluster.

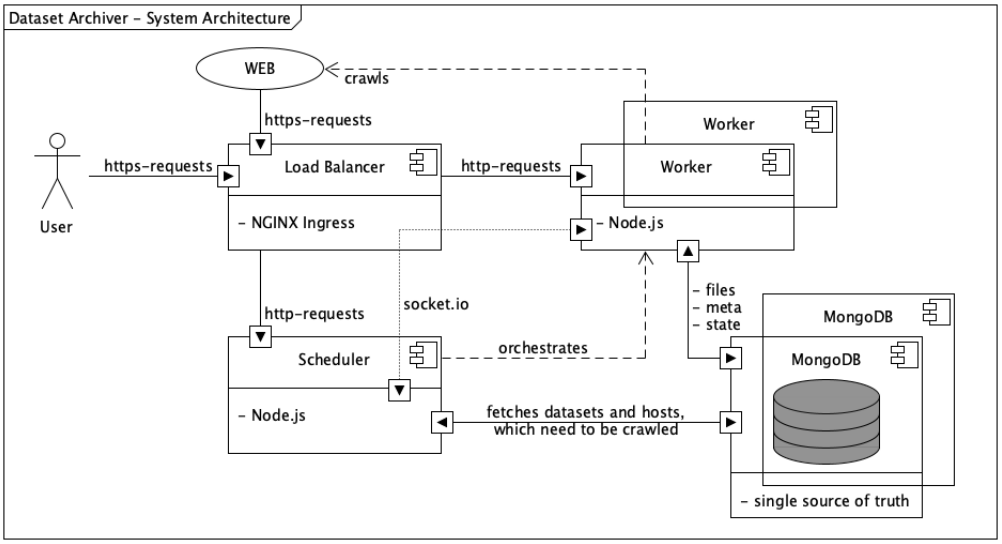
In order to scale the system, it is possible to plug in other clusters or add nodes to the cluster. This means we can dynamically add more resources and spread our workload. Our websocket software architecture further also allows users to act as crawling clients.

To further understand each of the components we provide architectural diagrams listed below.

4.1.1 System Structure

Figure 1 presents how a user can interact with the system and how the different parts are connected with each other.

Figure 1: Architecture



The MongoDB can directly be accessed by the scheduler software and the crawling clients. This crawling instances can be distributed over different clusters and work therefore in a parallelized way. The scheduling component can fetch datasets and hosts from the database which are ready to be crawled. It is further connected to the Ingress to orchestrate the clients.

If a user accesses the ODArchive, the NGINX Load Balancer distributes the requests in a Round-Robin fashioned way, which means the clients are ordered and the requests are cycled each after another. So if there are 3 clients (c) and 5 requests (r), the requests will be handled by the clients in following order:

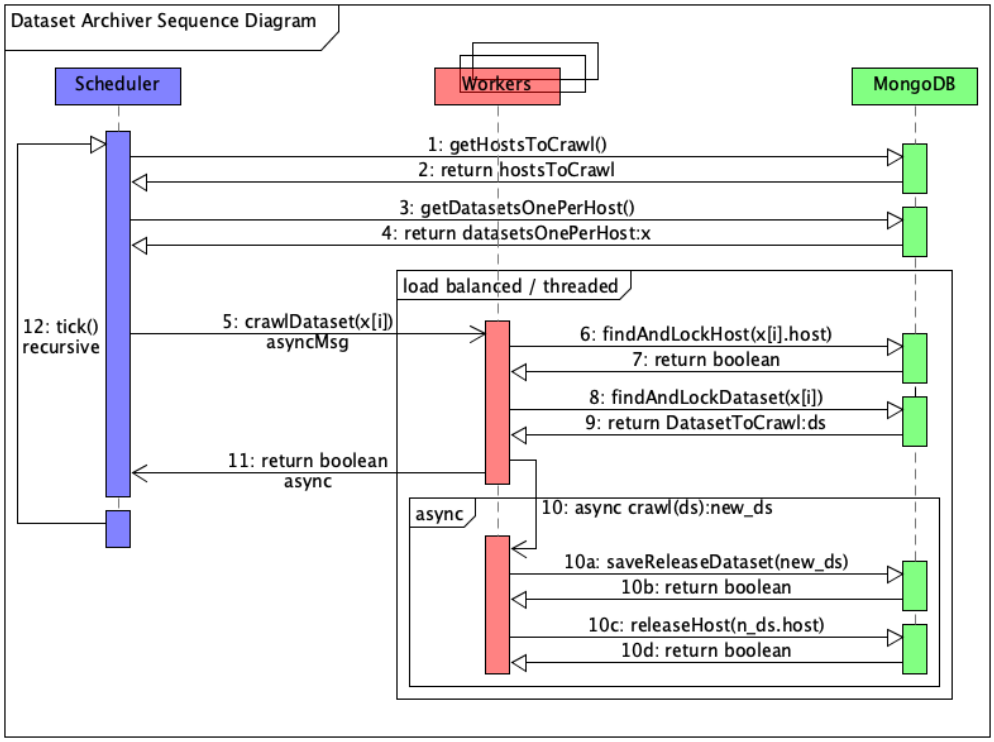
$$r1 \Rightarrow c1;\ r2 \Rightarrow c2;\ r3 \Rightarrow c3;\ r4 \Rightarrow c1;\ r5 \Rightarrow c2$$

The user is therefore also able to interact with the crawling clients via the NGINX Reverse Proxy and can access the API with http-requests or connect to the master via a socket.

4.1.2 Sequence Diagram

Figure 2 depicts the sequential interaction flow of each scheduling cycle between the three components Scheduler (S), Client (C) and MongoDB (DB). (The ⇒ symbol declares the direction of communication.)

Figure 2: Sequence Diagram



1. [S ⇒DB]: Database query to get all hosts which are free to crawl.
2. [DB ⇒ S]: Database returns all hosts which are free to crawl.
3. [S ⇒DB]: Database query to get all datasets which are free to crawl.
4. [DB ⇒ S]: Database returns all datasets which are free to crawl.
5. [S ⇒ C]: Asynchronous call to a client to crawl a dataset.
6. [C ⇒DB]: The client tries to lock the host, to ensure no other client can crawl the

same host simultaneously.

7. [*DB* ⇒ *C*]: Returns either true or false, depending on the hosts locking state.

8. [*C* ⇒*DB*]: The client tries to lock the dataset, to ensure no other client can crawl the same dataset simultaneously.

9. [*DB* ⇒ *C*]: Returns either true or false, depending on the datasets locking state.

10. [*C* ⇒ *C*]: The client starts an asynchronous crawl of the requested dataset, if the locking was successful.

10a. [*C* ⇒*DB*]: After crawling, the client saves and releases the currently crawled dataset.

10b. [*DB* ⇒ *C*]: Returns either true or false, depending on the datasets locking state.

10c. [*C* ⇒*DB*]: The client also releases the host to be able to get crawled again.

10d. [*DB* ⇒ *C*]: Returns either true or false, depending on the datasets locking state.

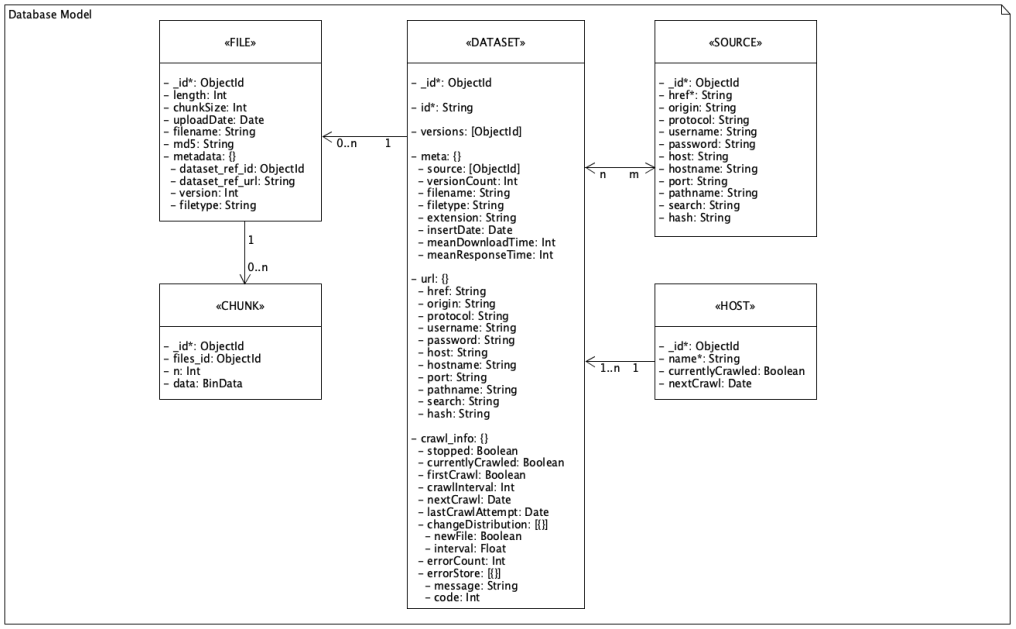
11. [*C* ⇒ *S*]: The client either returns true or false, depending on the state of the asynchronous crawl.

12. [*S* ⇒ *S*]: The Scheduler then recursively starts the next cycle.

4.1.3 Database Model

The Database Model in figure 3 provides a detailed overview of the attributes each collection in the database consists of and the relation between this collections.

Figure 3: Database Model



DATASETs store the essential information for our crawler to work. They consist of two unique identifiers, *_id* and *id*, an array of versions and the three objects *meta, url* and *crawl_info*.

FILEs hold all information about the individual versions of the datasets referenced by their ids in the versions array from the datasets collection.

CHUNKs store the actual data of the files and reference the files collection with its *_id* in the *files_id* field.

HOSTs are necessary for our locking mechanism and the host politeness to work properly like the *currentlyCrawled* field which is a boolean value.

SOURCEs are referenced by the source array field in the meta object of the datasets collection. One dataset can therefore have multiple sources and one source can be referenced by multiple datasets as well.

4.2 Data Access & Client Interface

Application Programming Interface (API) We provide the following API endpoints to interact with the Dataset Archiver. The API is devided into a publicly available API for searching and retrieving our crawled OD resources and a private API used for maintanance, requiring resp. credentials.

Detailed usage examples of the different APIs are documented on our Webpage at `https://archiver.ai.wu.ac.at/api-doc`.

4.2.1 Public API

`/stats/basic` Basic statistics on the data stored in the crawler's database.

`/get/dataset/{URL}` Returns a JSON object of a dataset description by its referencing URL.

`/get/datasets/{domain}` Returns a JSON object of all dataset descriptions provided by the same domain.

`/get/dataset/type/{TYPE}` Returns a JSON object of all dataset descriptions which offer resources with the specified filetype e.g. 'text/csv' or just 'csv'.

`/get/file/{URL}` Returns a resource (crawled file) by its referencingURL (i.e., for `dc:accessURL`s the latest downloaded version is retrieved, or, resp. a concrete `ds:hasVersion` URL can be provided directly).

`/get/files/type/{TYPE}` Returns a zip file containing all versions of the specified filetype e.g. 'text/csv' or just 'csv'.

/get/files/sparql?q={QUERY} Returns a zip file of the resource versions specified by a SPARQL query, that is, all the files corresponding to (version or dataset) URLs that appear in the SPARQL query result cf. detailed explanations below.

4.2.2 Private API

/post/resource?secret=SECRET Adds a new resource to the crawler by posting a JSON object containing the URL of the resource, the URL of the portal and the format e.g. 'text/csv' or 'csv'. Only the URL of the resource is mandatory and a secret key credential is needed to post resources.

/post/resources?secret=SECRET Adds several resources at once in batch, using the same parameters as above.

/crawl?id=ID&domain=DOMAIN&secret=SECRET Tells the clients which resource has to be crawled. A crawl can be enforced with this endpoint.

4.2.3 SPARQL Endpoint

```
PREFIX arc: <https://archiver.ai.wu.ac.at/ns/csvw#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX csvw: <http://www.w3.org/ns/csvw#>
PREFIX dcat: <http://www.w3.org/ns/dcat#>
PREFIX dc: <http://purl.org/dc/elements/1.1>
INSERT {
  <https://offenedaten.de/dataset/be8c1bf6-50cf-4fab-8ea3-179ca947652a>
     dcat:accessURL <https://www.berlin.de/daten/liste-der-kfz-kennzeichen/kfz-kennz-d.csv> .
  <https://www.berlin.de/daten/liste-der-kfz-kennzeichen/kfz-kennz-d.csv>
     dcat:mediaType "text/csv" ;
     dc:title "kfz-kennz-d.csv" ;
     dc:hasVersion <https://archiver.ai.wu.ac.at/api/v1/get/file/id/5e863ee2b511a4001191dcf8_0> ;
     dc:hasVersion <https://archiver.ai.wu.ac.at/api/v1/get/file/id/5e863ee2b511a4001191dcf8_1> .
  <https://archiver.ai.wu.ac.at/api/v1/get/file/id/5e863ee2b511a4001191dcf8_0>
     dc:identifier "0eec56f69acbda76b375ee982dbd4d7e" ;
     dc:issued "2020-04-06T22:09:56.336Z" ;
     dcat:byteSize 12642 .
  <https://archiver.ai.wu.ac.at/api/v1/get/file/id/5e863ee2b511a4001191dcf8_1>
     dc:identifier "74f78308cb653142663c057744cde84b" ;
     dc:issued "2020-04-12T22:09:56.336Z" ;
     dcat:byteSize 12642 . }
```

Figure 4: Example `INSERT` statement to add the dataset meta-information.

We make the metadata of the collected and archived datasets queryable over SPARQL by providing the corresponding meta-information in a triple store; the endpoint is available at https://archiver.ai.wu.ac.at/sparql.

To describe the datasets we make use of the Data Catalog vocabulary (DCAT) [28] for all crawled datasets (`dcat:Datast`) to specify links to the

portal (`dcat:Catalog`) where datasets were published, as well as `dcat:accessURL`s of *resources* and their respective format (`dcat:mediaType`).

Additionally, for tabular data resources, we provide metadata using the CSV on the Web vocabulary (CSVW) [29]: CSVW provides table-specific properties, such as `csvw:tableSchema` and `csvw:datatype`s per column. Section 4.2.3 shows an example of the meta-information stored for an archived dataset.

In this case, as the dataset is a CSV, we also insert CSVWeb metadata as shown in Section 4.2.3. See Section 4.4 details about the meta-data generation.

```
INSERT {
  _:csv csvw:url <https://archiver.ai.wu.ac.at/api/v1/get/file/id/5e863ee2b511a4001191dcf8_0> ;
    arc:rows 403 ;
    arc:columns 3 .
  _:csv csvw:dialect [
    csvw:encoding "utf-8" ;
    csvw:delimiter "," ;
    csvw:header true ] .
  _:csv csvw:tableSchema [
    csvw:column <https://archiver.ai.wu.ac.at/api/v1/get/file/id/5e863ee2b511a4001191dcf8_0#1> ;
    csvw:column <https://archiver.ai.wu.ac.at/api/v1/get/file/id/5e863ee2b511a4001191dcf8_0#2> ] .
  <https://archiver.ai.wu.ac.at/api/v1/get/file/id/5e863ee2b511a4001191dcf8_0#1>
    csvw:name "Stadt bzw. Landkreis" ;
    csvw:datatype "string" ;
    rdfs:range <http://dbpedia.org/ontology/Place> .
  <https://archiver.ai.wu.ac.at/api/v1/get/file/id/5e863ee2b511a4001191dcf8_0#2>
    csvw:name "Bundesland" ;
    csvw:datatype "string" ;
    rdfs:range <http://dbpedia.org/ontology/PopulatedPlace> . }
```

Figure 5: `INSERT` statement of example CSV meta-information.

Data Download via SPARQL Besides the APIs to directly access files from our crawl and the SPARQL interface to query metadata, we also offer a way of directly downloading data parameterized by SPARQL queries, i.e., for queries that include any URLs from the subject (*datasetURL*) or object (*versionURL*) of the `dc:hasVersion` property in our triple store, we provide a direct, zipped, download of the data: here *versionURL*s will directly refer to specific dowloaded file versions, whereas any *datasetURL* will retrieve the resp. latest available version in our corpus.

For instance, the query in Figure 6 selects all archived resources from a specific *data portal* (`data.gv.at`),[9] collected after a certain *time stamp*, with a specific *HTTP media type* (in this case CSV files); executing this query

[9]To filter datasets by certain data portals we enriched the descriptions by information collected in the Portal Watch (`https://data.wu.ac.at/portalwatch/`): we use

```
SELECT ?versionURL WHERE {
  ?datasetURL arc:hasPortal ?Portal ;  # ?datasetURL: the download URL of a specific resource
                                       # ?Portal: a dcat:catalog indexed in Portal Watch
    dc:hasVersion ?versionURL ;        # ?versionURL: a crawled version of the resource
    dcat:mediaType ?mediaType .        # ?mediaType: media type as per HTTP response.
  ?versionURL dc:issued ?dateVersion . # ?dateVersion: crawl time.

  FILTER (?Portal = <http://data.gv.at> &&
          ?mediaType = "text/csv"       &&
          strdt(?dateVersion, xsd:dateTimeStamp) >= "2020-05-10T00:00Z"^^xsd:dateTimeStamp) }
```

Figure 6: Example query to get a set of URLs of archived datasets.

at our SPARQL user interface (https://archiver.ai.wu.ac.at/yasgui) gives an additional option to retrieve the specific matching versions directly as a zip file. Alternatively, given this query to the `/get/files/sparql?q={QUERY}` API mentioned above, will retrieve these without the need to use the UI.

4.3 Traffic and Workload-management

To ensure our system does not overstrain our hosts and their underlying network infrastructure, we implemented some strategies to distribute the workload. We also ensured that our system easily scales to meet the needs of crawling large corpora.

4.3.1 Parallelization and Scalability

In order to parallelize our workload not only the Kubernetes infrastructure provides us with load balancing the work over the client pods, also the database can be sharded over more than one node.

Parallelization Although JavaScript is neither threaded nor does it use multiprocessing, we can mimic this with utilizing containerized software and distributing the load over mutliple pods in the container. Also the asynchronous programming style of JavaScript increases the amount of concurrent crawls by a single client pod.

Scalability There are two methods for addressing scalability of a system: vertical and horizontal scaling. The first is limited by the machine operating the system. Therefore we are focusing on sharding our MongoDB across multiple machines in a horizontal manner. This will lead to higher throughput

`arc:hasPortal` to add this reference. More sophisticated federated queries could be formulated by including the Portal Watch endpoint [2] which contains additional metadata.

operations and with this we can handle very large and a great amount of datasets.[30]

Each shard contains a subset of the data and each query is routed by a MongoDBs "mongos" instance, providing an interface between client applications and the sharded cluster. A shard key defines which node stores file chunks: we shard by dataset id plus the version number as shard key to keep all chunks of single files on the same node. The combination of Ingress, Kubernetes and MongoDB connected through micro-services can by extended dynamically, by adding more nodes, when needed.[30]

4.3.2 Dynamic crawl frequency

Umbrich, Mrzelj, and Polleres [6] propose to implement the scheduler as an adaptive component,

> "... which determines the next crawl time for URLs based on the content change information and current download frequencies."

Therefore we considered using the comparison sampling method developed by Neumaier and Umbrich [12] and also take into account the Nyquist sampling theorem.[31] This theorem states,

> "... that an analog signal waveform may be uniquely and precisely reconstructed from samples taken of the waveform at equal time intervals, provided the sampling rate is equal to, or greater than, twice the highest significant frequency in the analog signal."

This means, to recreate a frequency from unknown source, the sampling rate must at minimum be twice as high as the frequency itself.

```
let currInterval = (now - lastCrawlAttempt) // 5000
let changeDistribution = [
  {newFile: true, interval: 5000},
  {newFile: false, interval: 4600},
  {newFile: false, interval: 6200},
  {newFile: true, interval: 3800},
  {newFile: false, interval: currInterval}]

// produces: [15800, 8800]
let intervalOfNewFiles =
    changeDistribution.reduce((acc, curr) => {
      if (curr.newFile == true || acc.length == 0) {
        acc.push(curr.interval)
      } else {
```

```
15          acc[acc.length - 1] += curr.interval
16        }
17        return acc;
18      }, []);
19  // produces: 12300
20  let interval = calcAvarage(intervalOfNewFiles)
21  //devide crawl time by 2 if it has changed
22  if (versionHasChanged) interval /= 2
23
24  setNextCrawlAttempt(interval)
```

Listing 3: Function to calculate next crawling attempt

To compare the samples we store the distribution of changes back in time for a fixed amount of entries. The example in 3 shows how we compare the last five crawling attempts. We then calculate the mean interval in which the dataset has changed. If the dataset has changed, we divide the newly proposed interval by two as the Nyquist sampling theorem advises, to ensure that our sampling rate is twice as high as the changerate.

In Listing 3, the five "changeDistribution" samples are reduced to the intervals between new files. Then the average change rate is calculated and if the file has changed, the change rate is divided by two. Afterwards the next crawl attempt is being set.

We also implemented a minimum and maximum interval for our datasets to be getting crawled.

4.4 Data Management

While implementing and testing, we decided to only store files with a file size less than 100 Megabytes. Our crawler counts the downloaded bytes and if the maximum is reached, it cancels the download and drops the already downloaded data. We are also trying to analyse the data we crawl on the fly and MongoDB enforces compression over the saved chunks.

4.4.1 Type Detection and Data analysis

In order to detect the file-type we use the NPM package 'stream-file-type'[10] which provides a way to determine the file-type while piping the request to our MongoDB. It does so by checking the first 4100 bytes of a file. This is very efficient and saves an enormous amount of memory.

[10] https://www.npmjs.com/package/stream-file-type

For further content based analysis we currently only support CSVs by utilizing the 'csvengine' microserver from the Data Portal Watch.[2] It is located at https://archiver.ai.wu.ac.at/csvprofiler/api/v1/ and with this service we can populate our sparql endpoint with information about the contents of a given CSV. It heuristically detects the encoding, delimiters, as well as column datatypes of a CSV table, and provides this information using the `csvw:dialect` property. It further tries to detect if the CSV provides a header row, to extract column labels.

4.4.2 Compression

The WiredTiger Storage Engine provides four different compression options available when using MongoDB 4.2. These options can be varied on each collection to be able to optimize the storage recreation cost for each collection differently.

For our use case we decided to compare the default snappy option with the new zstd method. We used sample data of 17,25 GB with the snappy option enabled and compared this sample data to the zstd compression method, which just used 16,05 GB. This 7% reduction in file size will help reduce the needed storage even further.

4.4.3 Resource Handling

When someone adds resources to our system, we need to compare all his entries with our matching sources. Therefore we use the "python-shell"[11] NPM package to also execute Python Scripts parallel to our JavaScript code. To accomplish this, we exchange JSON data via the standard input.

```
1  obj = json.loads(input())
2  resp = {}
3  for oldDS in obj["old"]:
4    for newDS in obj["new"]:
5      if (len(newDS['meta']['source']) > 0 and
6        oldDS['id'] == newDS['id']):
7          IDs = list(set(oldDS['meta']['source']) | set(
               newDS['meta']['source']))
8
9          #just add if new sources
10         if (len(IDs) > len(oldDS['meta']['source'])):
11           #search for key to have unique entries
12           if oldDS['_id'] in resp:
```

[11] https://www.npmjs.com/package/python-shell

```
13              #union on unique href key
14              combIDs = list(set(resp[oldDS['_id']]) | set(
                    IDs))
15              #just add if new sources
16              if (len(combIDs) > len(resp[oldDS['_id']])):
17                resp[oldDS['_id']] = combIDs
18            #first key entry
19            else:
20              resp[oldDS['_id']] = IDs
21
22  print(json.dumps(resp))
```

Listing 4: Datasetsource comparison with Python

To compare each newly added resource with an existing dataset resource, we export the data to be analyzed by a Python Script because this deep comparison can be handled by Python in a more efficient way. In Listing 4 we demonstrate how all source entries of an existing dataset are compared to newly added source entries.

4.5 Dependencies and Open Issues

For our code to work properly we rely on different dependencies. Here we list our core dependencies we are using from the NPM package registry:

- csv-parser
- express
- helmet
- mime
- mongodb
- mongoose
- mongoose-unique-validator
- node-fetch
- python-shell
- robots-parser
- socket.io
- socket.io-client
- stream-file-type

Open Issues Although our system performs very well and runs stable, we need to improve our politeness metrics and the compression rate of the MongoDB GridFS storage. Also the metadata generation has to be enhanced and more unit-tests have to be programmed to advance further development.

Another big problem by now is the limitation of our host politeness rules. To address this issue, an intelligent strategy to determine the concurrent

requests a host is able to serve is needed. This could potentially be a research question on its own.

5 Findings

This sections describes the tests we conducted with our kubernetes cluster infrastructure and tries to give an impression of the capabilities of the crawler.

5.1 Corpus of the archivers database

At the moment of writing this work, the database of our crawler stores following numbers of documents in its collections, as table 1 shows:

Count	Collection
1.136.872	Datasets
7.746	Hosts
442.293	Sources
6.420.887	Files

Table 1: Collections

We also discovered that most datasets are hosted by data.opendatasoft.com. Table 2 shows the top-5 most indexed hosts:

Count	Host
66.389	data.opendatasoft.com
59.519	clss.nrcan.gc.ca
44.268	services.cuzk.cz
39.871	satc.rncan.gc.ca
27.656	www.geoportal.rlp.de

Table 2: Datasetcount per Host

By the time we finished writing this thesis our archiver has collected over 6.420.887 files from 1.136.872 URLs. We discovered that 8.59% of this files on disc are duplicates. Mainly because of saving error pages from portals which do not provide proper HTTP headers.

Overall we collected 7 TB worth of data, but with the compression of MongoDBs WiredTiger engine, our collection of chunks only consumes 4 TB of disc space. We limited our system to collect only files smaller than 100 Megabytes and Table 3 lists the distribution of the kilobyte-size of our collected files.

Quantile of Files	Kilobyte
10 %	< 3
25 %	< 13
50 %	< 54
75 %	< 204
85 %	< 504
95 %	< 3.204
99 %	< 27.638

Table 3: File Size Distribution

5.2 Monitoring and Bench-marking

To analyze the overall performance of our system, we conducted a 15 minutes crawling test-run. Therefore we used a cleaned database with 0 crawling attempts before. The database only consisted of the datasets, hosts and sources listed in table 1.

The node in our cluster for this experiment provided 25.27 GB of RAM and 8 CPU cores. We spawned 6 Client Pods, one Master Scheduler and one MongoDB Instance on this single node. Overall we used around 80% of RAM, of which 75% where used by the MongoDB. Unfortunately the CPU load was at 100% because all of our components where running on the same node.

Table 4 gives us an overview of the scheduled datasets to download for this 15 minutes. Our scheduler wanted to crawl 37.959 datasets but only 21.988 were started because the Clients can have a maximal amount of asynchronous crawling attempts to adjust the load a client is capable of handling. Of this 21.988 datasets, 9.965 were aborted due to bad status codes or because they reached the maximal file size. At the end we successfully saved 12.023 files and added 9.965 error messages to our database.

Datasets	Status
37.959	wanted to crawl
21.988	started to crawl
9.965	aborted
12.023	downloaded

Table 4: Status

In table 5 we list the avarage file size of the downloaded data, the total amount of data we have and how much physical storage is needed for the whole crawling system.

Size	Description
1,358 Mb	average file size
16,327 Gb	total files stored
12 Gb	physical storage needed

Table 5: Storage

In table 6 we provide a distribution of different data types we have crawled in just 15 minutes and overall.

15 min Count	File Type	overall	File Type
4.687	html	253.688	text/html
1.743	xml	145.121	application/json
1.321	csv	121.175	text/csv
1.130	pdf	98.383	application/zip
789	json	94.595	application/xml
599	zip	48.185	application/pdf
486	bin	46.396	text/xml
405	xlsx	27.641	application/xlsx
222	msi	23.624	application/octet-stream
133	txt	21.725	application/x-msi

Table 6: File Type Distribution

We also measured the time it takes for a dataset to be downloaded. On average a file needs 22,089 seconds to be stored in our system, which can be seen in table 7.

Time	Description
15,952 sec	average response time
6,137 sec	saving time
22,089 sec	total download time

Table 7: Time Stats

We further analyzed the network throughput of the system and recognized, that our Grafana metric tool only provides the overall network statistics. This means, that also the connection to our database is counted as network traffic. Therefore we subtracted the average throughput of our Clients from the average throughput of the MongoDB and realized that our system produces 31,6 MB/s of traffic on average, which are 252,8 Megabits per second.

Speed	Component
65,4 MB/s	Clients
33,8 MB/s	MongoDB
31,6 MB/s	difference

Table 8: Network Traffic

This numbers are just to showcase the capabilities of this system. In a real world application the system would schedule the crawling attempts over time in a dynamic way. This means there might not be so much system load because not all datasets would need to be crawled at the exact same time.

6 Conclusion and Further Research

Conclusion In this thesis we have developed a system capable of downloading huge amounts of data in a parallelized, scalable and polite way. ODArchive is set to provide easy access to a large, up-to-date corpus of datasets from OD portals: we archive regularly re-crawled versions of underlying data resources for datasets from these portals, based on an adaptive, heuristically estimated crawl rate and have presented a scalable extensible infrastructure to sustainably run such an archive.

In our resource track paper for ISWC2020 we show how our system can be used by linking tabular OD datasets to existing KGs as well as interlinking them amongst each other by finding reference tables within the corpus. The initial results of this paper clearly suggest that the characteristics of the structured data found on OD portals and readily provided in our corpus are quite different from other available copora, such as Web Tables. Additionally, as our framework keeps on running, it shall also enable temporal analyses over the evolution of OD resources.

We also evaluated the scalability of our system by indexing datasets from 137 OD portals over a period of 8 weeks:

It supports (i) increasing resources of the Kubernetes cluster, (ii) connecting other clusters and balancing/distributing the work load across different nodes, and (iii) horizontal scaling by including additional MongoDB shards, if storage needs to be increased.

We provide a query and user interface to interact with the system: Users are now also able to retrieve information about the data corpus via an API and a SPARQL endpoint, which allow flexible filtering, and generation of sub-corpora for further reproducible experiments.

By the time of writing this thesis our ODArchive has collected over

6.420.887 files from 1.136.872 URLs totalling 7 TB worth of data. Our main conclusions can be summarized as follows:

- We discovered that all files on disc consume just 4 TB of disk space and about 8.59% of them are duplicates.
- Our analysis showed that 50% of these files are smaller than 55 Kilobytes, 95% are smaller than 3,2 Megabytes and 99% are smaller than 27,6 Megabytes.
- In total 169.123 URLs retrieved from the OD portals where not available at all: either the HTTP response code indicated that the dataset is not retrievable, or the host was not even reachable.
- Another issue that we encountered is that many of the dataset URLs return HTML pages, which might not be the actual data we requested. In total there are 253.679 html files of which around 25% are returned with an error code. This means of all the 169.123 datasets with error codes, 37,68% are html files. In future work we want to develop heuristics to detect if the datasets could be retrieved using accurate content negotiation.
- The average downloadtime while crawling a version is 3966,83 milliseconds or 3,96683 seconds and on average a dataset consists of 7.978 versions.

Further Research To reduce the storage cost of our system, further research needs to be done for de-duplicating the now generated chunks of files.

We are also keen to generate more meta date while downloading and further enhance the politeness rules of the crawling system. One approach could be to determine a politeness metric based on the number of datatsets a host serves.

In order to extend the corpus, we are also planing to include datasets from other sources, e.g. data science platforms such as Kaggle, or research data platforms such as Zenodo or Harvard Dataverse.

Another interesting field of research might now be to analyze the changes of the downloaded files and produce indexes to query the crawled data based on the changes over time. We are also looking into detecting reference tables in our corpus and try to further provide semantic labelling of our semi-structured data.

Also the combination with already existing systems of the Vienna University of Economics and Business like the newly introduced JupyterHub[12] or the Learn@WU[13] platform might be considered to motivate the implementation of the previously mentioned secondary requirements. This would dramatically enhance the services our system is able to provide.

In future work we plan to also analyze and attempt to interlink other structured formats in our corpus; additionally, as our framework keeps on running, it shall also enable temporal analyses over the evolution of OD resources. The infrastructure shall allow detailed analyses overall, but also with a narrower scope, restricting to data from particular portals or regions. Last, but not least, we invite everybody to use ODArchive and provide feedback (e.g., in terms of additional API feature requests).

Availability The ODArchive can be accessed at:
`https://archiver.ai.wu.ac.at/`.
The source code is available under the MIT license on GitHub:
`https://github.com/websi96/datasetarchiver`.

7 Acknowledgements

We thank our System Administrator Martin Beno for the help setting up the Kubernetes cluster and also the Department for Information Systems of the Vienna University of Economics and Business for providing us with the needed technological resources. We also thank ao.Univ.Prof. Dr Johann Mitlöhner for jointly writing the resource track paper for ISWC2020.
As of writing my Bachelor Thesis, I personally thank Dr. Sebastian Neumaier and Univ.Prof. Dr. Axel Polleres for guiding me through the process of writing this work.

[12] `https://jupyter.ai.wu.ac.at/`
[13] `https://learn.wu.ac.at/`

References

[1] Thomas Weber et al. "ODArchive – Creating an archive for structured data from Open Data Portals". In: 19th International Semantic Web Conference (ISWC 2020), Lecture Notes in Computer Science (LNCS), Virtual Conference (Athens, Greece), Nov. 2020. URL: https://aic.ai.wu.ac.at/~polleres/publications/webe-etal-2020ISWC.pdf.

[2] Sebastian Neumaier, Jürgen Umbrich, and Axel Polleres. "Automated quality assessment of metadata across open data portals". In: *Journal of Data and Information Quality (JDIQ)* 8.1 (2016), pp. 1–29. ISSN: 19361955. DOI: 10.1145/2964909. URL: https://aic.ai.wu.ac.at/~polleres/publications/neum-etal-2016JDIQ.pdf.

[3] Dan Brickley, Matthew Burgess, and Natasha F. Noy. "Google Dataset Search: Building a search engine for datasets in an open Web ecosystem". In: *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 2019, pp. 1365–1375. DOI: 10.1145/3308558.3313685.

[4] Ramanathan V Guha, Dan Brickley, and Steve Macbeth. "Schema. org: evolution of structured data on the web". In: *Communications of the ACM* 59.2 (2016), pp. 44–51.

[5] Oliver Lehmberg et al. "A large public corpus of web tables containing time and context metadata". In: *Proceedings of the 25th International Conference Companion on World Wide Web*. 2016, pp. 75–76.

[6] Jürgen Umbrich, Nina Mrzelj, and Axel Polleres. "Towards capturing and preserving changes on the Web of data". In: *CEUR Workshop Proceedings*. 2015. URL: https://pdfs.semanticscholar.org/971b/178200a0bc14735116ace49a0b164e68a926.pdf.

[7] Amol Deshpande. *Why Git and SVN Fail at Managing Dataset Versions*. 2015. URL: http://www.cs.umd.edu/~amol/DBGroup/2015/06/26/datahub.html (visited on 05/15/2019).

[8] Stephen Cook. "The P versus NP problem". In: *The millennium prize problems* (2006), pp. 87–104. URL: http://www.claymath.org/library/monographs/MPPc.pdf.

[9] Rolf Sint et al. "Combining unstructured, fully structured and semi-structured information in semantic wikis". In: *CEUR Workshop Proceedings*. Vol. 464. 2009, pp. 73–87.

[10] Patrick Oliver Riemer. "Implementing a "polite" proxy for different Web Crawling Use Cases". unpublished. 2017.

[11] Google Webmaster Central Blog. *A note on unsupported rules in robots.txt*. URL: https://webmasters.googleblog.com/2019/07/a-note-on-unsupported-rules-in-robotstxt.html (visited on 11/13/2019).

[12] Sebastian Neumaier and Jurgen Umbrich. "Measures for assessing the data freshness in open data portals". In: *Proceedings - 2016 2nd International Conference on Open and Big Data, OBD 2016*. 2016. ISBN: 9781509040544. DOI: 10.1109/OBD.2016.10.

[13] archive.org. *The Internet Archive*. URL: https://archive.org/about/ (visited on 06/19/2020).

[14] Common Crawl. *Common Crawl*. URL: https://commoncrawl.org/about/ (visited on 06/19/2020).

[15] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. "Profiling relational data: a survey". In: *VLDB J.* 24.4 (2015), pp. 557–581. DOI: 10.1007/s00778-015-0389-y.

[16] Kaggle. *Kaggle*. URL: https://www.kaggle.com/ (visited on 06/19/2020).

[17] Javier D. Fernández, Patrik Schneider, and Jürgen Umbrich. "The DBpedia Wayback Machine". In: *Proceedings of the 11th International Conference on Semantic Systems*. SEMANTICS 15. Vienna, Austria: Association for Computing Machinery, 2015, pp. 192–195. ISBN: 9781450334624. DOI: 10.1145/2814864.2814889. URL: https://doi.org/10.1145/2814864.2814889.

[18] Souvik Bhattacherjee et al. "Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff". In: *PVLDB* 8.12 (2015), pp. 1346–1357. DOI: 10.14778/2824032.2824035. URL: http://www.vldb.org/pvldb/vol8/p1346-bhattacherjee.pdf.

[19] GitHub.com. *Versioning large files*. URL: https://help.github.com/en/articles/versioning-large-files (visited on 05/28/2019).

[20] Anant P. Bhardwaj et al. "DataHub: Collaborative Data Science & Dataset Version Management at Scale". In: *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. 2015. URL: http://cidrdb.org/cidr2015/Papers/CIDR15%5C_Paper18.pdf.

[21] postgresql.org. *Large Objects*. URL: https://www.postgresql.org/docs/current/lo-intro.html (visited on 05/28/2019).

[22] Daniel Coupal and Ken W. Alger; mongoDB.com. *Building with Patterns: The Document Versioning Pattern*. URL: https://www.mongodb.com/blog/post/building-with-patterns-the-document-versioning-pattern (visited on 05/28/2019).

[23] nodejs.org. *Overview of Blocking vs Non-Blocking*. URL: https://nodejs.org/de/docs/guides/blocking-vs-non-blocking/#concurrency-and-throughput (visited on 12/11/2019).

[24] nodejs.org. *Node.js*. URL: https://nodejs.org/en (visited on 11/13/2019).

[25] kubernetes.io. *Kubernetes*. URL: https://kubernetes.io/ (visited on 11/16/2019).

[26] Johann Mitloehner et al. "Characteristics of open data CSV files". In: *Proceedings - 2016 2nd International Conference on Open and Big Data, OBD 2016*. 2016. ISBN: 9781509040544. DOI: 10.1109/OBD.2016.18.

[27] F. Doglio. *Pro REST API Development with Node.js*. Apress, 2015. ISBN: 9781484209172. URL: https://books.google.at/books?id=kjUwCgAAQBAJ.

[28] Fadi Maali and John Erickson. *Data Catalog Vocabulary (DCAT)*. W3C Recommendation. W3C Recommendation. http://www.w3.org/TR/vocab-dcat/. Jan. 2014.

[29] Rufus Pollock et al. *Metadata Vocabulary for Tabular Data*. W3C Recommendation. https://www.w3.org/TR/2015/REC-tabular-metadata-20151217/. Dec. 2015.

[30] mongoDB.com. *Sharding*. URL: https://docs.mongodb.com/manual/sharding/ (visited on 11/14/2019).

[31] Martin H. Weik. "Nyquist theorem". In: *Computer Science and Communications Dictionary*. Boston, MA: Springer US, 2001, pp. 1127–1127. ISBN: 978-1-4020-0613-5. DOI: 10.1007/1-4020-0613-6_12654. URL: https://doi.org/10.1007/1-4020-0613-6_12654.