TU WIEN Informatics

Persistente Identifikation und Referenzierung von sich ändernden Daten in Computerexperimenten

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Florian Wörister, BSc
Matrikelnummer 1126205

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
Mitwirkung: Projektass. Dr.techn. Mag. Tomasz Miksa

Wien, 15. Oktober 2020

Florian Wörister	Andreas Rauber

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Persistent Identification and Referencing of Evolving Research Data in Computational Experiments

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Florian Wörister, BSc
Registration Number 1126205

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
Assistance: Projektass. Dr.techn. Mag. Tomasz Miksa

Vienna, 15th October, 2020

Florian Wörister | Andreas Rauber

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Florian Wörister, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. Oktober 2020

Florian Wörister

Danksagung

An erster Stelle möchte ich mich bei meinen Eltern bedanken, die mir dieses Studium ermöglicht haben.

Auch bei meinen Betreuern Adreas Rauber und Tomasz Miksa möchte ich mich sehr herzlich für die großartige Unterstützung und die vielen konstruktiven Arbeitsgespräche bedanken.

Kurzfassung

Die Rolle computergestützter Experiment als Quelle von wissenschaftlichen Erkenntnissen gewinnt zunehmend an Bedeutung in der wissenschaftlichen Community. Dabei sind die verwendeten Datensätze oft nicht statisch, sondern entwickeln sich im zeitlichen Verlauf weiter (z.B. werden Einträge hinzugefügt, modifiziert oder gelöscht). Beim Publizieren der Ergebnisse ist es üblich den Sourcecode in einem entsprechenden Repository und die benutzten Daten auf der Institutswebseite zu veröffentlichen. Auf die verwendeten Datensätze wird meist via URL verwiesen. Da diese jedoch ihre Gültigkeit verlieren können, führt das zu nicht reproduzierbaren Forschungsergebnissen. Diese gängige Praxis macht es anderen Forschern schwer, publizierte Experimente zu finden und wiederzuverwenden. In dieser Arbeit präsentieren wir ein Plugin für ein Forschungsdatenrepository, welches Wissenschaftlern ermöglicht Subsets von sich ändernden Datensätzen zu veröffentlichen und zu zitieren. Zusätzlich wird eine Schnittstelle bereitgestellt, die es computergestützten Experimenten durch Angabe des entsprechenden persistenten Identifikators ermöglicht, die Datensätze zu beziehen. Es wird gezeigt, dass die persistente Identifikation von Sourcecode und verwendeten Daten die FAIRness dieser Experimente erhöht. Als Resultat werden fünf Guidlines definiert, um Wissenschaftler dabei zu unterstützen, die FAIRness ihrer veröffentlichten Experimente zu erhöhen.

Die Implementierung ist auf GitHub verfügbar:
`https://doi.org/10.5281/zenodo.4015614`

Abstract

The role of computational experiments as a source of insights is becoming increasingly important in the scientific community. The datasets used for experiments are often not a static resource but rather evolve over time (e.g. records are added because new information is available, existing records are modified or deleted in order to correct mistakes). When it comes to publishing the results of those experiments, it is common practice to put the code on an institutional website or into a code repository and reference the dataset by providing a link. As links can break, this leads to a lack of reproducibility. Therefore, it is cumbersome for other researchers to find and re-use computational experiment assets (code or data) if everyone stores them on their institutional website. In this thesis, we provide a new plugin for the a data repository which enables researchers to publish and cite specific subsets of evolving datasets. In addition, we provide an interface that can be used by computational experiments to persistently retrieve datasets from the repository by providing their persistent identifier. Furthermore, we show how persistent identification of experiment source code and data subsets can increase the FAIRness of the published experiment. As a result, we define five guidelines to support researchers in increasing the FAIRness of their computational experiments based on subsets of continuously evolving data.

The implementation is available on GitHub:
`https://doi.org/10.5281/zenodo.4015614`

Contents

CHAPTER 1

Introduction

1.1 Motivation

The emergence of information technology had a big impact on the way research is done. In earlier scientific paradigms, researchers tried to gain new knowledge by observing nature and coming up with theoretical models to describe the world. Later on, the availability of computers allowed scientists to conduct experiments by simulating complex phenomena. As claimed by Jim Gray, science is moving towards a data-intensive paradigm [1]. Instead of observing the world, scientists are now analyzing large datasets and performing computational experiments based on those datasets.

This paradigm comes with new challenges, e.g. reproducing the outcomes of computer based experiments.When in 2016 a survey asked 1,576 scientists if they think there is a reproducibility crisis in the research community, 90% replied that, in their opinion, there is at least a slight crisis[2]. Several reasons leading to this outcome were identified, among them the unavailability of original lab data, applied methods, and source code.

In 2016, Camerer et al. conducted an investigation on reproducibility of laboratory experiments in economic science [3]. Only in 11 out of 18 studies, they found the same significant effect in the same direction. Even in regards to reproducible experiments the authors stated that for some studies a re-execution of the experiments was challenging. In the following subsections, we give two motivating examples that demonstrate the issues with current practices of publishing computational experiments and the data they are based on.

1.1.1 Example 1: The Reinhart and Rogoff Error

In 2010, Reinhart and Rogoff, two of Harvard's most influential academic economists, claimed in a seminal paper that average real economic growth declines when a country's debt exceeds 90% of the gross domestic product [4]. Three years later, a student from

the University of Massachusetts Amherst tried to replicate their findings by re-executing the calculations described in the paper but failed to reproduce the published results[5]. Herndon, Ash, and Pollin discovered that due to computation errors the conclusion of the original paper was incorrect. Besides the impact on Reinhart's and Rogoff's reputations, this error also influenced political decisions which led to high unemployment during the financial crisis of 2009 in Europe [6].

The following three errors were identified[5]:

- An unusual weighting of the countries was used.
- Some of the data were selectively excluded (e.g. growth periods in New Zealand while the country's dept was above the 90% threshold).
- Due to a coding error, entire countries were excluded from the calculations (e.g. Austria).

Although the full datasets were available[1] on the authors' website, it is remarkable that it took over two years to identify these errors. Some errors could not be identified until the original working spreadsheets of Reinhart and Rogoff were published.

Notwithstanding these accusations, the authors did not distance themselves from their findings, but rather tried to justify their results, e.g. confronted with the issue of unusual gaps within the used datasets they responded:

"*The "gaps" are explained by the fact there were still gaps in our public data debt set at the time of this paper, a data set no one else had ever been able to construct before and which we now have filled in much more completely.*" [7]

One can thus conclude that the dataset used in the experiment has changed over time. Therefore, in order to reconstruct the results, it is not only important to know which dataset (and applied filters) was utilized by the experiment, but also to know the exact version of the dataset used for the experiment's initial execution.

1.1.2 Example 2: Peer-Assisted Information-Centric Network (PICN)

This motivating example discusses the paper "Peer-Assisted Information-Centric Network (PICN): A Backward Compatible Solution", which was published in the IEEE Digital Library in 2017[8]. It introduced a new architecture for information-centric computer networks. The evaluation of the suggested approach was partly done by conducting a trace driven simulation.

To examine the reproducibility of this simulation, we tried to re-execute the simulation experiment.

[1] `http://www.carmenreinhart.com/data/browse-by-topic/` (visited on: 10/13/2020)

```
read log files...
Traceback (most recent call last):
  File "EventsPreprocessor.py", line 335, in <module>
    generate_berkeley_events_file(path, alexa_rtt_bw)
  File "EventsPreprocessor.py", line 197, in generate_berkeley_events_file
    if fields[13]=='GET':
IndexError: list index out of range
Traceback (most recent call last):
  File "SimulatorExecution.py", line 90, in <module>
    trace_day=trace_day)
  File "SimulatorExecution.py", line 20, in __init__
    self.simulator.loadEvents(True)
  File "/code/SimulatorGenerator.py", line 153, in loadEvents
    fevents = open(self.events_folder + 'events_info.txt')
IOError: [Errno 2] No such file or directory: '../data/events_dir/
    events_info.txt'
```

Figure 1.1: Code Ocean Stacktrace of the PICN simulation re-execution.

Since the source code was already provided in a Code Ocean Capsule[2], we assumed the re-execution of the experiment would be straightforward. Triggering a re-execution in the capsule, however, failed (the stacktrace of the execution is depicted in 1.1). The failure was caused by the preprocessing script trying to parse the LICENCE file which lead to the `IndexError`. Removing the file from the data folder solved this issue.

The experiments were based on two datasets, the *UC Berkeley Home IP Web Traces* and the *IRCache 2007 trace files*. For both, hyperlinks were provided to retrieve the full datasets, but the links were broken by the time this thesis was written. Only for the *UC Berkeley Home IP Web Traces*, a smaller subset was included in the source code project. Regarding the second dataset, the authors left a comment in the README file: "*But unfortunately the download link is no more available. So please contact me at z.zali@ec.iut.ac.ir if you want the files.*"[3] A request for this dataset stayed unanswered[4].

By using the Wayback Machine[5], we were able to retrieve the UC Berkeley dataset, but there are deviations between the characteristics of the dataset described in the publication and the actual retrieved dataset. The retrieved dataset consists of 9,244,728 requests from 8,374 unique clients, whereas in the publication the authors described the dataset as a collection of 8,559,556 request from 8,377 unique clients. The paper states in the summary table of the dataset's characteristic that the requests were tracked over a period of four days. In contrast, the website of the original dataset states that the requests were recorded between the 1st and the 19th of November 1996.

[2]DOI: `https://doi.org/10.24433/CO.d553b1ad-81dd-4152-8bd6-0165b78edad7` (visited on: 10/13/2020)
[3]DOI: `https://doi.org/10.24433/CO.d553b1ad-81dd-4152-8bd6-0165b78edad7`, path: /code/README.md (visited on: 10/13/2020)
[4]The mail was sent on the 1st of April, 2020
[5]`https://archive.org/web/` (visited on: 10/13/2020)

A closer look at the published source code led to the conclusion that the authors only described the characteristics of their pre-processed datasets, which only included `GET` requests that also retrieved a response.

This case makes it clearer that, although the authors tried to provide a reproducible experiment along with the publication, there is still a need for a solution that enables researchers to publish computational experiments and the datasets they are based on. This case also shows that due to limited disk space it is not always possible to publish the full dataset along with the experiment source code.

1.2 Problem Statement and Research Questions

As already elaborated in the motivating examples in Section 1.1, a major issue of current computational experiments is a lack of verifiability due to missing or insufficient research data management.

Even when research data is published along with the scientific results, it is frequently only stored on the authors' websites, e.g. the datasets from Reinhart and Rogoff, or referenced using a URL which can break over time, as we have seen in the case of Zali et al.'s publication. For the publication of Herndon, Ash, and Pollin, simply a zip file containing the experiment code and the datasets was uploaded to the university's website. While this allows access to the source code and datasets, it would make finding and subsequently reusing research assets rather cumbersome if all researchers were to follow this example.

Therefore, this thesis aims to tackle the following research questions:

Q1: To what extent can we automate persistent identification of data for continuously evolving databases used in scientific experiments?

- How can continuously evolving data be stored in a versioned way in order to retrieve older states of the dataset?
- What components need to be added to a research data repository to be capable of storing the required data for re-execution of historic queries in a reproducible way?
- To what extent does the versioning overhead affect the queries on the current state of the resource in terms of query run time?
- To what extent is the query run time affected by the number of records stored in a repository resource?
- To what extent is the time for retrieving persistently identified subsets affected by the number of updates that are performed after the persistent identifier was issued?

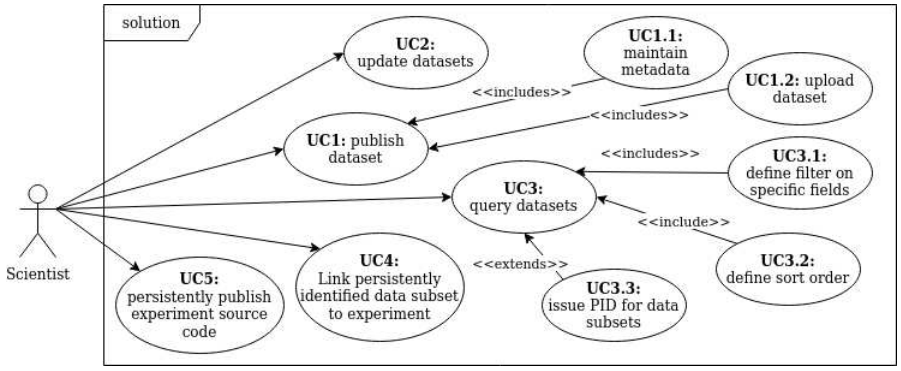


Figure 1.2: Use cases of a scienticst wanting to publish a computational experiment.

Q2: How can FAIRness of computational experiments based on continuously evolving data be increased?

- How can persistently identified data subsets be published in a CKAN repository?
- How can persistently identified Git repository versions be published in a CKAN repository?

1.3 Aim of this Thesis

In order to solve the problems described in Section 1.2, the goal of this thesis is to introduce a system that supports researchers in publishing research datasets and enables linking of data subsets to computational experiments.

The use cases we aim to support are depicted in the two diagrams. Figure 1.2 shows the use cases of scientists that want to publish datasets and use subsets of published datasets in computational experiments. This involves uploading the data, updating the dataset and its metadata, and generating subsets of the data by submitting queries to the system. An extended case of the query use case is issuing a persistent identifier for queries that were submitted to the repository. These identifiers are used to persistently retrieve the exact same subset, e.g. in computational experiments.

Figure 1.3 depicts the retrival of the exact same source code and data subset (including metadata describing the dataset) on which a certain published experiment is based.

Furthermore, a good solution should not only scale in terms of increasing dataset sizes but also in number of changes committed to a dataset, i.e. performing updates on the dataset should not slow down the retrieval of a subset for which a persistent identifier was already issued. Otherwise an identification can not be guaranteed in the long term as the retrieval of the dataset may become infeasible when the amount of updates reach a certain level.

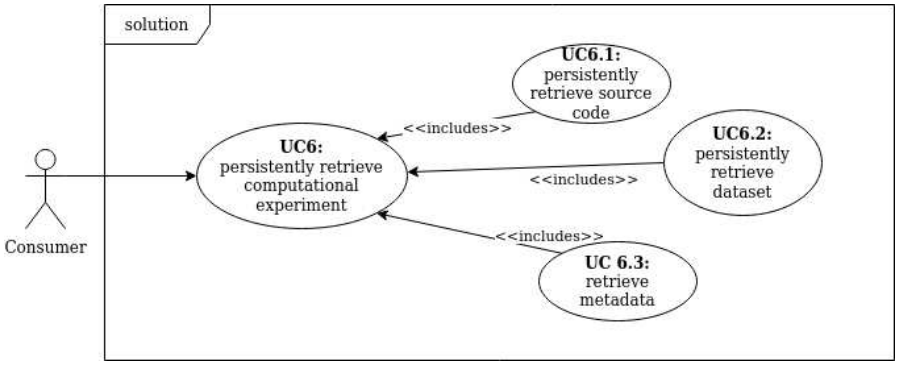


Figure 1.3: Use cases of a consumer wanting to retrieve source code and underlying data of published experiments.

1.4 Thesis Structure

The chapters of this thesis are structured as follows:

Chapter 2 gives an introduction to the principles of data citation, as well as an overview of existing prototype implementations.

Chapter 3 provides an overview of the proposed conceptual design. In addition, architectural design decisions are discussed.

Chapter 4 describes in detail, how the goals outlined in Section 1.3 can be achieved by implementing the conceptual design described in Chapter 3.

Chapter 5 first gives a description of the test setup that was used to evaluate the proposed implementation, and is then concluded by the evaluation results.

Chapter 6 summarizes the findings of this thesis.

CHAPTER 2

Related Work

The first two sections of this chapter cover the efforts already made to improve research data management by defining guidelines, principles, and recommendations on how to publish (2.1) and cite (2.2) research data. Subsequently, the state-of-the-art of implementing research data management services are discussed. This includes current platform solutions (e.g. CKAN) and already deployed research data repositories (e.g. Eurostat, data.ccca, etc.). Finally, we discuss the state-of-the-art in source code versioning, including examples of source code versioning solutions for source code of computation experiments.

2.1 The FAIR Guiding Principles

As stated by Roche et al. in 2015, there is a need for improvement in public data archiving in terms of reusability [9]. Due to the fact that the amount of research data produced is increasing, there is a need to define main criteria for good data management.

In 2014, a workshop held in Leiden (Netherlands) with the title "Jointly Designing a Data Fairport" tackled those problems and resulted in the development of four guiding principles for improving the reusability of published scientific data, namely the *FAIR Guiding Principles* [10].

These guidelines suggest that research objects should meet the following criteria [10]:

1. **Findable**

   F1 (meta)data are assigned a globally unique and persistent identifier

   F2 data are described with rich metadata (defined by R1 below)

   F3 metadata clearly and explicitly include the identifier of the data it describes

   F4 (meta)data are registered or indexed in a searchable resource

2. **Accessible**

   A1 (meta)data are retrievable by their identifier using a standardized communications protocol

      A1.1 the protocol is open, free, and universally implementable

      A1.2 the protocol allows for an authentication and authorization procedure, where necessary

   A2 metadata are accessible, even when the data are no longer available

3. **Interoperable**

   I1 (meta)data use a formal, accessible, shared, and broadly applicable language for knowledge representation.

   I2 (meta)data use vocabularies that follow FAIR principles

   I3 (meta)data include qualified references to other (meta)data

4. **Reusable**

   R1 meta(data) are richly described with a plurality of accurate and relevant attributes

   R1.1 (meta)data are released with a clear and accessible data usage license

   R1.2 (meta)data are associated with detailed provenance

   R1.3 (meta)data meet domain-relevant community standards

A goal of this thesis is to foster reusability of datasets in the scientific community by providing a suitable repository solution. As the FAIR principles give suggestions on how to publish datasets in a way that they can be reused by other scientists, the suggested solution described in the subsequent chapters relies on these principles.

2.2 Data Citation

Researchers include bibliographic citations in their publications to give credit to the findings on which their works is based. This practice helps not only to prevent scientific fraud and plagiarism, but also allows peers to reproduce and verify the results.

Due to the already discussed shift towards a data intensive paradigm in the scientific community, it became necessary to also cite the datasets on which published results are based.

2.2.1 The Evolution of Data Citation

The idea of citing data already existed in 1979 when Dodd suggested that data should be seen as "intellectual work" as well, and therefore be bibliographically cited like literature [11]. The evolution of data citation since 1979 can be described in four phases [12].

During the first phase, awareness was raised that datasets which are stored in archives should be considered as intellectual work similar to published literature, followed by a phase where the practice of citing datasets in publications emerged. This also included the utilization of persistent identifiers (e.g. DOIs) to reference the datasets. Although the ability to verify and reproduce research results was always a motivation for archiving data that supported scientific findings, in the third phase the focus was on the rising importance of citing data in the context of reproducibility of research results. In the current phase, effort is put into integrating data citation practices into the scholarly ecosystem.

2.2.2 Initiatives

In the past, several committees have tried to define and standardize how data citation should be done in order to provide guidelines for the scientific community.

Effort has been made to define principles and guidelines to establish data citation in the research community, e.g. the Committee on Data for Science and Technology (CODATA) released a report on Data Citation Standards and Practices in 2013 [13]. Another attempt to define principles of data citation was made by the Future Of Research Communications and E-Scholarship (FORCE11)[1] community which was published in the "Amsterdam Manifesto on Data Citation Principles".

But as stated by Crosas in “The Evolution of Data Citation: From Principles to Implementation”, until the publication of the *Joint Declaration of Data Citation Principles* there was no set of principles that aligned with all so far existing guidelines [12]. Through collaboration between all organizations that already came up with data citation principles (e.g. CODATA, FORCE11), seven principles were identified that aimed to unify the previously existing principles:

1. **Importance:** Data should be considered legitimate, citable products of research. Data citations should be accorded the same importance in the scholarly record as citations of other research objects, such as publications.

2. **Credit and Attribution:** Data citations should facilitate giving scholarly credit and normative and legal attribution to all contributors to the data, recognizing that a single style or mechanism of attribution may not be applicable to all data.

3. **Evidence:** In scholarly literature, whenever and wherever a claim relies upon data, the corresponding data should be cited.

[1] `https://www.force11.org` (visited on: 10/13/2020)

4. **Unique Identification:** A data citation should include a persistent method for identification that is machine actionable, globally unique, and widely used by a community.

5. **Access:** Data citations should facilitate access to the data themselves and to such associated metadata, documentation, code, and other materials, as are necessary for both humans and machines to make informed use of the referenced data.

6. **Persistence:** Unique identifiers, and metadata describing the data, and its disposition, should persist – even beyond the lifespan of the data they describe.

7. **Specificity and Verifiability:** Data citations should facilitate identification of, access to, and verification of the specific data that support a claim. Citations or citation metadata should include information about provenance and fixity sufficient to facilitate verfiying that the specific timeslice, version and/or granular portion of data retrieved subsequently is the same as was originally cited.

8. **Interoperability and Flexibility:** Data citation methods should be sufficiently flexible to accommodate the variant practices among communities, but should not differ so much that they compromise interoperability of data citation practices across communities. [2]

2.2.3 Data Identification

So far citing data was described as listing references to the used datasets in a scientific publication, but in order to reference a dataset the referenced asset has to be identifiable. This means that a reader of a publication should be able to find the exact same dataset as it was used when the experiment was initially conducted. Unlike for immutable datasets, this can be a tedious task for dynamic dataset where records are added, modified, or deleted over time. The motivating example provided in Section 1.1.1 showed that improper citation of dynamic datasets leads to a lack of reproducibility of the publication.

Crosas identified three questions that emerge when dynamic data have to be identified:

- *The equivalence question.* How does one determine whether two data objects, not bitwise identical, are semantically equivalent (interchangeable for scientific computation and analysis)?

- *The versioning question.* How does one unambiguously assign, at the time of citation, a 'version' to a data object, such that someone referencing the citation later can retrieve or recreate the data object in the same state that it was at the time of citation?

[2] Data Citation Synthesis Group: Joint Declaration of Data Citation Principles. Martone M. (ed.) San Diego CA: FORCE11; 2014 [https://www.force11.org/group/joint-declaration-data-citation-principles-final].

- *The granularity question.* How does one unambiguously describe components and/or subsets of a data object for purposes of computations, provenance, and attribution? How does one incorporate this granularity with a bibliographic data citation to create a "deep" citation? [12] [13]

As the data identifications questions of Crosas are addressed by the RDA Recommendation for Data Citation, this thesis contributes to the scholarly ecosystem by demonstrating how an already existing data repository solution can be extended in a way that it aligns with these recommendations. This will result in a repository solution that is capable of citing and identifying subsets of evolving datasets.

2.2.4 Persistent Identifiers

As a common practice, web resources are referenced by their location, i.e. their Uniform Resource Locator (URL). The issue with referencing resources by providing their URL is, that the location of the resources can change (e.g. the domain of the server that hosts the file changes). This means that without any strategy, links can easily break over time which makes it hard to find those resources.

Unlike URLs, a Uniform Resource Identifier (URI) describes a resource's name, location or both. The idea to prevent broken links is add a redirection layer, that resolves URIs into a URL that describes where to find a specific resource. This concept is depicted in Figure 2.1. Physical as well as digital resources can be referenced with a URI. In addition to just referencing the digital object itself, the URI resolver can also point to related metadata and associated services.

There exist already several solutions for the problem of providing persistent identifiers on computer networks. Figure 2.2 gives an overview of which PID system was used by how many repository in December 2015. The following subsection discusses the three most common systems.

Handle

Handle is a distributed information system that provides a global name service for identifying digital networks on a computer network (e.g. on the internet). The first paper describing the system was written over the period of November 1993 to May 1995 by Robert Kahn and Robert Wilensky. The first implementation is operational on the internet since fall 1994[16].

In the context of Handle systems, a name is a combination of a prefix and a suffix which are separated by a slash. The prefix refers to the resource's namespace and the suffix is the resource name.

The system is organized hierarchically with one global naming service (the Global Handle Registry) on top. The naming services on the lower levels are referred to as Local Handle Services. Each LHS maintains its unambiguous namespace (e.g. a university has its

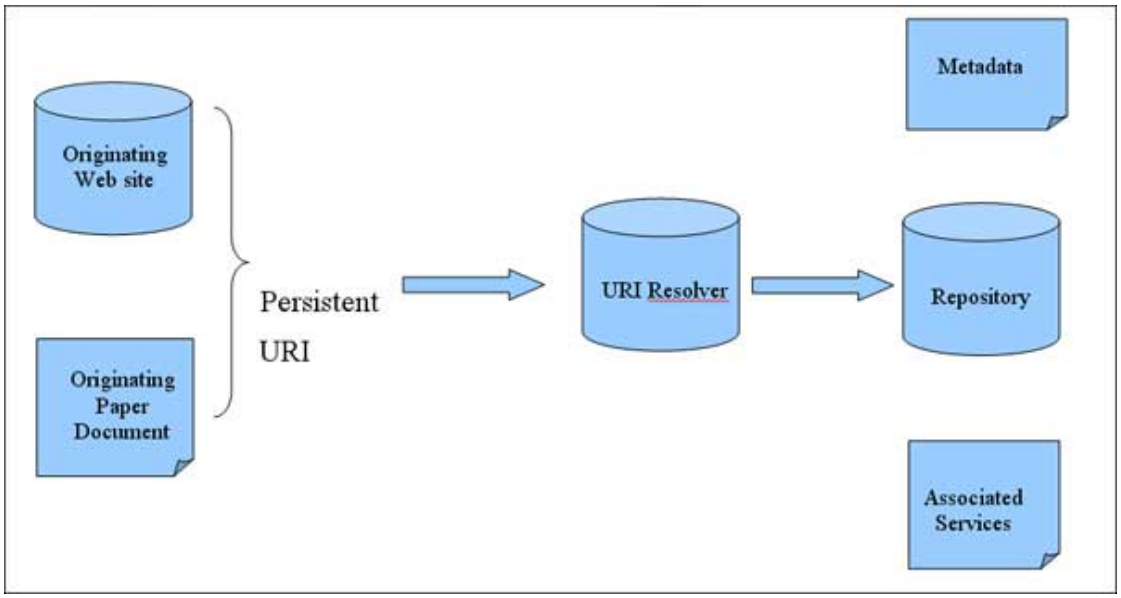


Figure 2.1: The functionality of a persistent URI resolver [14]

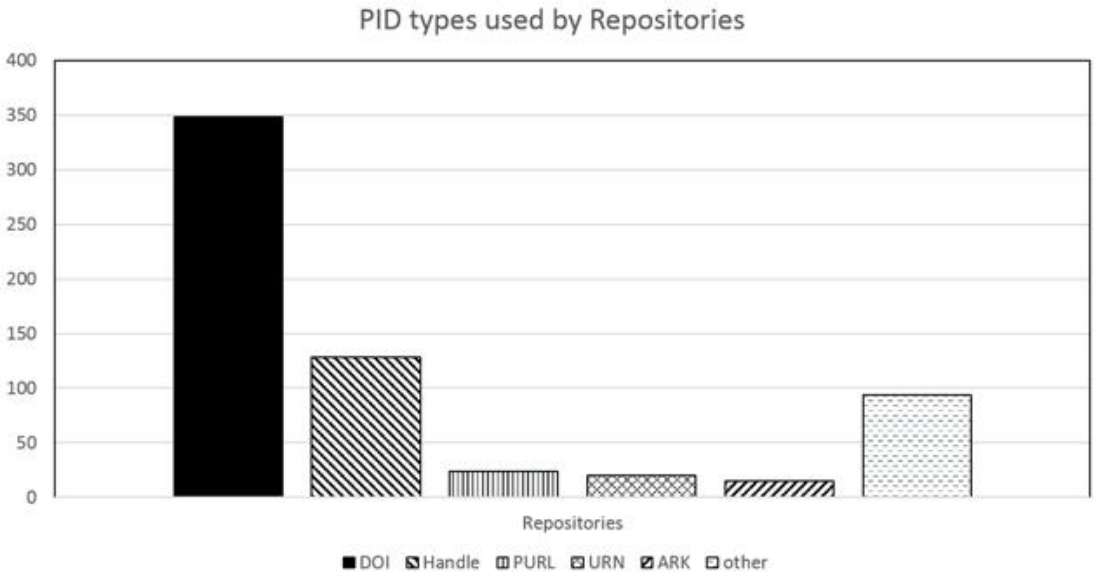


Figure 2.2: Number repositories using one of the listed PID systems [15].

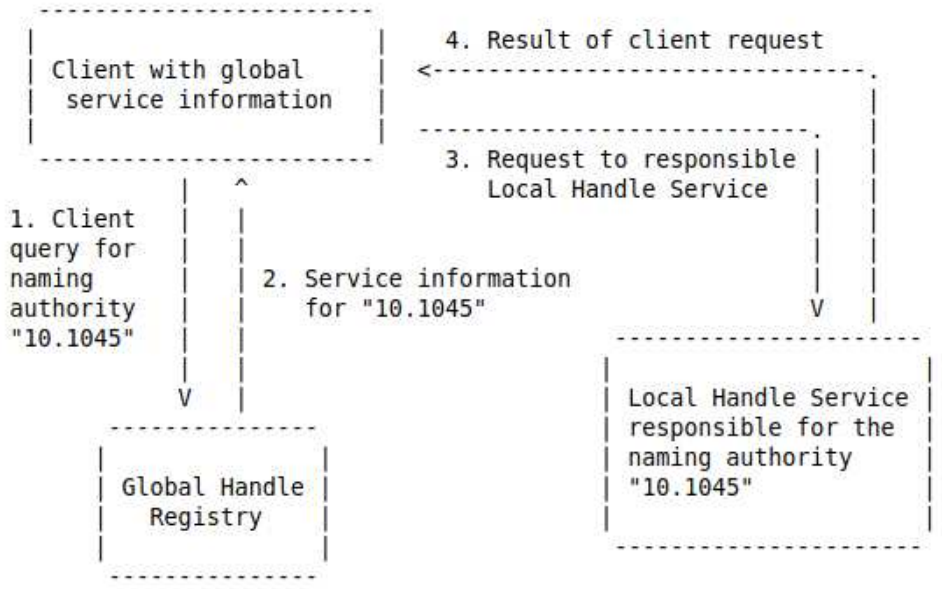


Figure 2.3: Handle name resolution process [17]

own namespace that is maintained by their LHS). The GHR itself only maintains the service information of its child LHS, i.e. if a client resolves a Handle, it first looks up the LHS information in the GHR. Subsequently, the PID is resolved to the actual location of the digital object by sending a resolution request to the corresponding LHS. The whole process is depicted in Figure 2.3.

DOI

DOI is a naming service that is curated by the International DOI Foundation (IDF). Although originating for the publishing industry, it was planned as a generic naming service for digital objects. As the system is based on Handle, the CNRI is still a technical partner of the IDF [3]. Since 2010 the DOI system is standardized by the ISO [4].

DOI uses Handle as a basic infrastructure for resolving persistent identifiers, therefore the syntax of DOIs and Handles is the same. The difference between this two systems is that Handle is a technological solution for resolving identifiers to a specific location. On the other hand DOI uses this Handle infrastructure but adds features and services to provide a "persistent, semantically interoperable identification of intellectual property entities"[5].

[3] https://doi.org/10.1000/182 Chapter 1.2 (visited on: 10/13/2020)
[4] https://www.iso.org/standard/43506.html (visited on: 10/13/2020)
[5] https://www.doi.org/factsheets/DOIHandle.html (visited on: 10/13/2020)

PURL

Persistent URLs were introduced by the Online Computer Library Center[6] (OCLC) in 1995. They were originally based on a fork of the Apache HTTP Server[7]. Since 2016 the PURL resolver service was migrated to the Internet Archive, an non-profit organization advocating a free and open internet[8]. The migrated service is based on a newly created software and not on a previous implementation.

The identifiers consist of a root URL of a resolver service (e.g. *http://resolverservice.com*) and a unique name (e.g. *resource1*) of the resource identified by the PURL resulting in the PID *http://resolverservice.com/resource1*.

Similar to the previous solutions by resolving a persistent identifier the client browser is redirected to the target location of the digital object. By being able to update the correct target URL, the persistence of the identifier can be maintained.

Relevance of Persistent Identifiers for this Thesis

To meet the goals defined in Section 1.3, an infrastructure for resolving identifiers to the location of the data subsets they are referring to is needed. Using a PID system only does not solve the problem as the files this PIDs are pointing to can still be modified (e.g. a PID resolves to a file on a FTP server which can be edited any time by a person that has access to this file).

To summarize, while PIDs are part of the solution, they alone are not sufficient. Section 2.2.5 is going discuss how a system can be implemented to support persistent data identification for subsets of evolving datasets. The recommendations discussed in this section are of importance as they form the basis of the solution presented in this thesis.

2.2.5 RDA Recommendation for Data Citation

The Research Data Alliance (RDA)[9] is a no-profit organization that aims to foster open sharing of data between researchers across disciplines and countries. The Data Citation Working Group is part of the RDA and focuses on the citation of subsets of dynamic data. They published 14 Recommendations (R1-R14) for Data Citation [18].

Being able to cite subsets of evolving data requires a separate data (**R1 - R2**) and a query store(**R3 -R12**), that keeps track of all queries that were submitted to the data store. The data store needs to support data versioning that also keeps track of when certain actions were performed on the data store. The query store keeps track of all queries, and normalizes them in order to be capable of identifying semantically

[6] Initially the organization was founded as Ohio College Library Center but was renamed due to its expansion.

[7] `https://httpd.apache.org/` (visited on: 10/13/2020)

[8] `https://cdm15003.contentdm.oclc.org/digital/collection/p15003coll6/id/649` (visited on: 10/13/2020)

[9] `https://www.rd-alliance.org` (visited on: 10/13/2020)

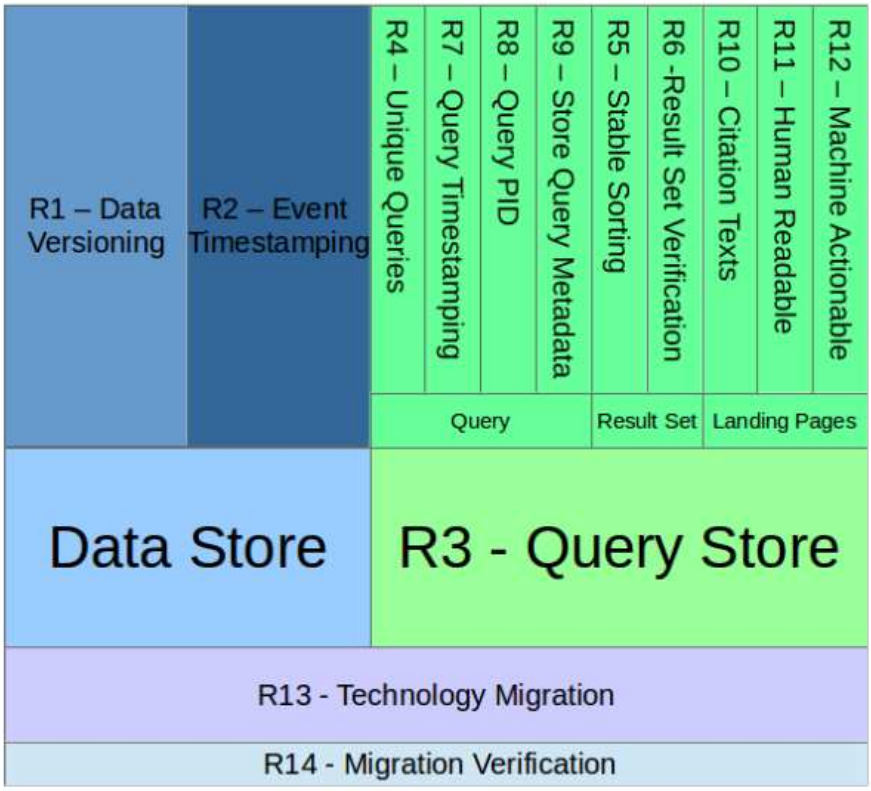


Figure 2.4: The Recommendations of Data Citation as Components [18]

equivalent queries (**R4**). In order to know the exact state of a datastore when a query was executed, for every query a timestamp is stored (**R7**). The requirements to the result set are that it guarantees stable sorting (**R5**) and that it is possible to verify the result set, e.g. by storing a hash value of the result set to every query (**R6**). **R10 - R12** suggest that there should be a machine actionable and human readable landing page that provides the retrieved subset, its metadata and a ready-to-use citation text. The last two recommendations cover organizational concerns in case of a technology migration. The relation between all recommendations is depicted in Figure 2.4.

This thesis contributes to the fourth phase of data citation (c.f. Section 2.2.1) by design and implementation of a data repository solution based on an already existing repository. This is done by extending the repository in a way that it aligns with the RDA Recommendations for Data Citation.

2.3 Research Data Management Services

Due to the ongoing shift towards data intensive science, the importance of managing and maintaining big amounts of data for research institutions is increasing. Furthermore,

Figure 2.5: Components of research data management support services [19]

the importance of making research data *FAIR* (as discussed in Section 2.1) results in a demand for research data repository solutions for higher educational institutes (HEIs). In the context of this thesis, data repositories refer to services that allow to store datasets along with its related meta data in a way that they are findable and accessible for other researchers.

In order to support HEIs in establishing a research data management service, Jones, Pryor, and Whyte developed a guide covering the main aspects that should be addressed by those institutions [19]. As depicted in Figure 2.5, implementing a research data management service involves several aspects. This thesis will focus on the implementation of a data repository.

Due to an increasing number of research data repositories, there is a need for a registry that keeps track of all available repositories and enables researchers to find a specific repository of interest. In 2012, the re3data.org project started to index all publicly available repositories[20].

The registered repositories are very heterogeneous in various aspects, e.g. the software on which they are based. Besides proprietary developed repository solutions, there also

exist several open source projects that provide ready-to-use software to setup a research data repository. The existing solutions are discussed in Section 2.3.1 in more detail. The present software solutions are the three most common open source repository solutions according to a comparison of research data management platforms done by Amorim et al. in 2017 [21].

On an organizational level, already existing repository solutions can be distinguished by the scientific domain they address, e.g. the *Alternative Fuels Data Center*[10] hosts data of a specific domain published by several institutions. On the other hand, there are repositories used to publish data of a specific institution like the *Eurostat*[11] repository which is curated by the statistical office of the European Union only. There are also general purpose services like *Zenodo*[12] which are not tied to any scientific domain. Additionally, these services are open to use for any researcher and therefore host data from a wide variety of institutions and scientific domains.

Section 2.3.2 discusses in depth already existing research data repository services to give an deeper understanding of how they emerged and what their advantages and disadvantages are. It covers not only the examples mentioned above, but also the *World Economic Outlook Database* as the data sets of this repositories were used as a data source for Reinhart and Rogoff's publication discussed in Section 1.1.1. In addition, this section will also cover the *data.ccca* repository as it provides the so far closest solution to the problem stated in Chapter 1.1.

2.3.1 Research Data Repository Software

Due to the demand for repository solutions, several projects were started to meet the requirements of current research data management. The solutions differ in availability of the source code, the existance of an API or the support of unique identifiers.

The following subsections will give an introduction to prevalent repository solutions.

EPrints

EPrints[13] is a free research data management tool developed at the University of Southampton. It was published in 2000 on GitHub[14] under GPL-3.0 licence. The repository was implemented in *Pearl*.

If used with its default configuration, EPrints serves as an open archive for research papers, but it is possible to use it as a repository for data (e.g. images, research data, audio archives) as well.

[10] `http://www.afdc.energy.gov/` (visited on: 10/13/2020)
[11] `https://ec.europa.eu/eurostat/de/data/database` (visited on: 10/13/2020)
[12] `https://zenodo.org` (visited on: 10/13/2020)
[13] `https://www.eprints.org` (visited on: 10/13/2020)
[14] `https://github.com/eprints` (visited on: 10/13/2020)

Since version 3.4, a flavor support was introduced that allows to setup the repository for a particular purpose. One of the offered flavors turns EPrints into a research data repository that supports data citation. Nevertheless, it does not align with the WGDC Recommendations for Data Citation.

In general, EPrints comes with fixed metadata fields, but it is possible to configure the available metadata fields on a per-repository basis.[15]

This software solution is already used by several institutions, e.g. University of Mannheim, Ludwig Maximilian University of Munich and University of Regensburg.

DSpace

DSpace[16] is a data repository solution mostly written in *Java*. Being a result of a cooperation between MIT Libraries and Hewlett-Packard (HP), the platform was released in 2002.

The repository is organized in items and collections, each item has one or more associated files and has to be part of a collection. The submission process is organized in several steps (represented by a separate UI page), e.g. a *describe*-step where the user has to provide metadata for the item that is going to be ingested. These steps are customizeable. Meta information is associated to items in the form of metadata fields which consist of a name(e.g. dc.title) and a value. An essential feature of DSpace is its capability of enforcing custom metadata schemas.

According to the DuraSpace Registry there are over 2000 DSpace installations in use, which makes it one of the most popular research data repositories.[17] A prominent example is the MIT Repository[18].

CKAN

CKAN[19] (**C**omprehensive **K**nowledge **A**rchive **N**etwork) is a repository solution that is maintained by the *Open Knowledge Foundation*. The OKF is a global non-profit organization that promotes the idea of open data with a focus on government information systems. Like for Eprints and DSpace the codebase is available on GitHub[20].

In CKAN, data assets organized in datasets include several resources (e.g. a csv file). Datasets belong to a group or an organization (e.g. TU Wien). Metadata can be added in the form of key-value pairs to either datasets or resources. Unlike DSpace, CKAN does not support the enforcement of a specific metadata schema. Examples of already

[15] `http://wiki.eprints.org/w/Metadata#Core_Properties` (visited on: 10/13/2020)
[16] `https://duraspace.org/dspace/` (visited on: 10/13/2020)
[17] `https://duraspace.org/registry/` (visited on: 10/13/2020)
[18] `https://mit.edu` (visited on: 10/13/2020)
[19] `https://ckan.org` (visited on: 10/13/2020)
[20] `https://github.com/ckan/ckan` (visited on: 10/13/2020)

used installations of this repository are *opendata.swiss*[21] or the European Data Portal[22]. In the context of this thesis the most relevant installation is the *data.ccca*[23] portal of the Climate Change Center Austria (CCCA).

2.3.2 Research Data Repository Services

Alternative Fuels Data Center

The AFDC is an example for a disciplinary repository as it hosts data of one scientific domain which is published by several institutions which are the *Office of Energy Efficiency & Renewable Energy*, *U.S. Department of Energy*, *U.S. Department of Transportation, Federal Highway Administration* and the *National Renewable Energy Laboratory, Clean Cities Program* [24].

It was launched in 1991 as a repository for performance data of alternative fuel vehicles but evolved over time to an online hub that not only provides data but also tools supporting transportation decision makers [25].

Besides the provided tools (e.g. interactive maps) the repository offers an API to access and download the datasets[26]. Nevertheless the provided data is not versioned in any sense and the downloaded datasets do not provide a persistent identifier that allows researchers to persistently refer to a specific dataset of this repository they used in their own experiments.

Eurostat

In the European Union, the statistical office (Eurostat) maintains a publicly accessible database, where statistics concerning the European Union are published. The data is collected by the member states who send them to Eurostat where the data is consolidated and made comparable before publication. As the database provides comprehensive statistics, it is possible to filter the data via a web interface as depicted in Figure 2.6

Although Eurostat supports versioning of datasets, there are no persistent identifiers assigned to the generated subsets.

Zenodo

In 2013, the European Organization for Nuclear Research (CERN) launched Zenodo[27], a general-purpose data repository, to support open science by enabling researchers to

[21] https://opendata.swiss/en/ (visited on: 10/13/2020)
[22] https://www.europeandataportal.eu/ (visited on: 10/13/2020)
[23] https://data.ccca.ac.at/ (visited on: 10/13/2020)
[24] re3data.org: Alternative Fuels Data Center; editing status 11/28/2017; re3data.org - Registry of Research Data Repositories. http://doi.org/10.17616/R3K303 (visited on: 10/13/2020)
[25] https://afdc.energy.gov/about.html (visited on: 10/13/2020)
[26] https://developer.nrel.gov/docs/transportation/ (visited on: 10/13/2020)
[27] re3data.org: Zenodo; editing status 2019-01-15; re3data.org - Registry of Research Data Repositories. http://doi.org/10.17616/R3QP53 (visited on: 10/13/2020)

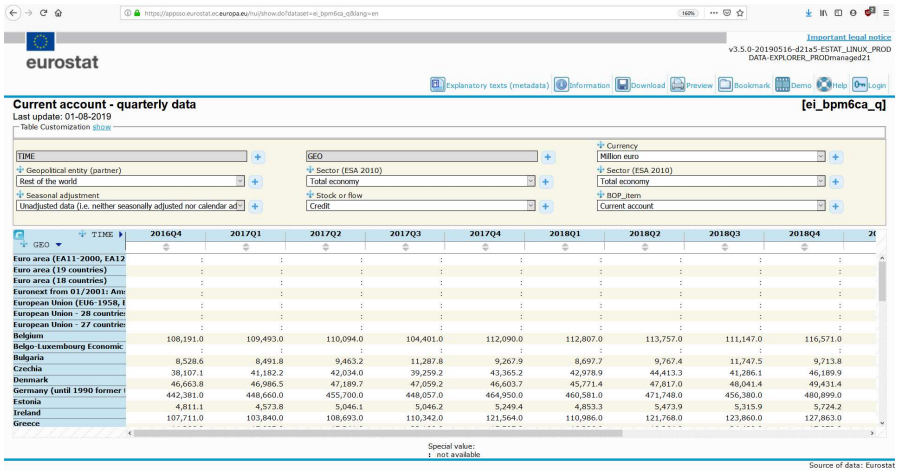


Figure 2.6: Filtering mask of the Eurostat database

publish their publications and datasets.

As for larger institutions the archival of datasets (e.g. data produced by the LHC at CERN) is often already handled by the institution itself, Zenodo tries to address the needs of smaller institutions and individual researchers that do not have the resources to archive and publish their research data.

The service itself is based on the *Invenio*[28] framework, which is also developed by CERN. Zenodo allows researchers to collaborate by curating community driven data collections. Another feature of Zenodo is that uploaded resources are versioned on a file level and to each version DOIs are assigned.

Although this allows to persitently identify a specific version of a dataset, this solution does not meet the requirements mentioned in Chapter 1.1 as it does not support generation and persistent identification of subsets of published datasets.

Zenodo also provides a GitHub integration which allows researchers to mint DOIs for specific versions of a software that is versioned by a GitHub repository. This already meets the requirements for being able to persistently identify a specific version of an experiments source code described in Section 1.3. For an productive environment this solution would be a valid solution but in this thesis a GitLab server and a Handle service (both hosted locally) are used to evaluate the suggested solution in order to prevent dependencies to external services.

World Economic Outlook Databases

The World Economic Outlook Database is a repository hosted by the International Monetary Fund. The website describes the database as follows:

[28] `https://invenio-software.org/` (visited on: 10/13/2020)

"The World Economic Outlook (WEO) database contains selected macroeconomic data series from the statistical appendix of the World Economic Outlook report, which presents the IMF staff's analysis and projections of economic developments at the global level, in major country groups and in many individual countries. The WEO is released in April and September/October each year."[29]

The database does not provide an API in order to access the data in a machine actionable way. The only way to retrieve datasets is via a web interface, where the results are not only displayed in HTML but can also be downloaded as CSV file. Furthermore, there are no persistent identifiers assigned to the datasets.

This database was one of the primary sources for the experiments conducted by Reinhart and Rogoff.

Open Data Portal Vienna

Another example of a CKAN installation is the Open Data Portal of Austria[30]. It is a single point of access for the open data portal of the European Union[31].

Although there exist plugins to make CKAN support PID assignment for its resources, this repository does not support persistent identification.

data.CCCA

The CCCA is an Austrian research network that coordinates climate research in Austria. It was founded in 2011 as a nonprofit organization and is supported by Austria's most important research institutions.[32] Their repository is based on CKAN and implements a feature to cite subsets of evolving NetCDF files. An overview of the implementation approach is given in Figure 2.7 [33].

Most of the recommendations are implemented in a CKAN extension, only data archiving and generation of subsets is done by a Thredds Data Server (TDS)[34].

This solution nearly covers all requirements needed for a solution of the problem described in Section 1.2. Nevertheless, the repository addresses the needs of a specific scientific domain, i.e. earth observation science. This also explains why subset generation is only supported for NetCDF files, which is a common file format in this domain.

[29] https://www.imf.org/external/pubs/ft/weo/2019/02/weodata/index.aspx (visited on: 10/13/2020))
[30] https://www.data.gv.at/ (visited on: 10/13/2020)
[31] https://www.europeandataportal.eu/ (visited on: 10/13/2020)
[32] https://ccca.ac.at/en/about-ccca/the-association-ccca (visited on: 10/13/2020)
[33] https://www.rd-alliance.org/system/files/documents/CCCA_DC_RDA_DynamicDCite_v3_inkl_manual.pdf (visited on: 10/13/2020)
[34] https://www.unidata.ucar.edu/software/thredds/current/tds (visited on: 10/13/2020)

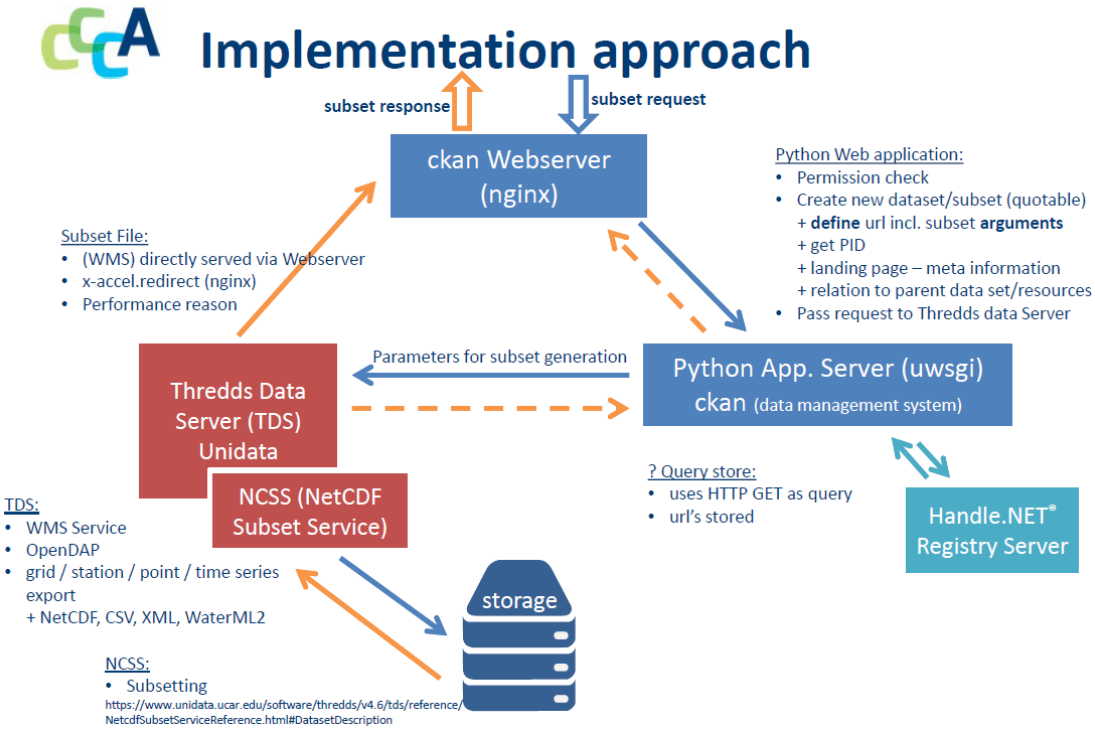


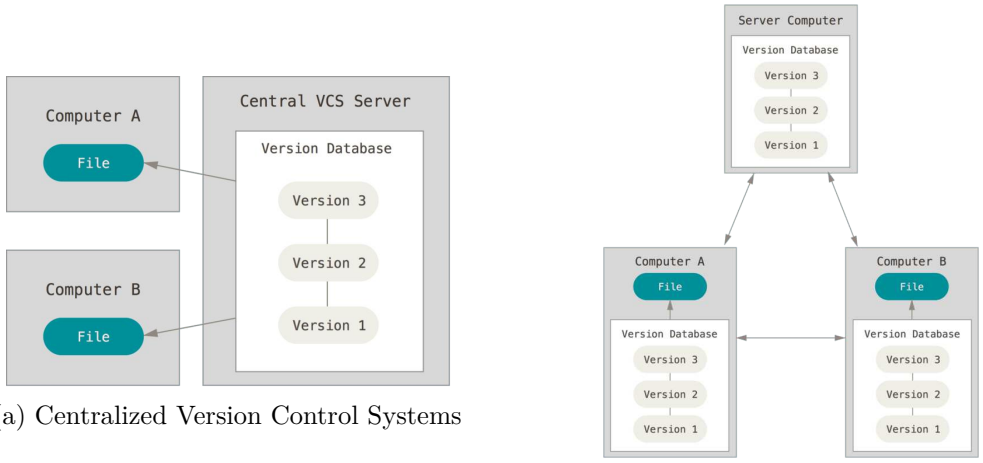
Figure 2.7: Implementation approach of the data.ccca repository

2.4 Software Versioning Control Systems

Since software is developed in teams and not by a single person, it is necessary to manage the process of source code creation. As described by Louridas, reasons for versioning source code are the fact, that developers want to go back to an older version, because the current version did not improve the solution, or the fact, that working in teams requires some solution in order to not interfere with changes made by team colleges [22].

In this thesis, source code versioning is utilized to improve provenance tracking of results from computational experiments on open datasets by publishing the source code of experiments in a version control system citing a specific version of the source code in publications.

There is already a variety of tools supporting developer teams versioning their source code. The most common solutions are discussed in the following subsections. A discussion on which version control system was used for this thesis is given in Section 3.1.3.



(a) Centralized Version Control Systems

(b) Distributed Version Control Systems

Figure 2.8: The conceptual differences between centralized and distributed version control systems [23]

2.4.1 Centralized Version Control Systems

Centralized Version Control Systems have a traditional server client architecture. Clients retrieve a specific (probably the current) version from the server and perform modifications on these so-called working copies and only work on these local copies instead of the centrally stored files.

One prominent implementation of a Centralized Version Control System is **Subversion**, which acts like an ordinary filesystem (clients can write/read files to/from it) with the difference that the central repository keeps track of every version that is stored on it, therefore any older version of a file or directory can be retrieved at any time.

Users commit file changes as commits to the central repository which decides if the commit is accepted or not (i.e. if there are conflicts, the commit will not be accepted). In case of a successful commit, all of the changes are applied to the repository and a new revision is issued, otherwise none of the changes are applied (therefore the commit can be seen as an atomic operation).

For referencing a specific version of, e.g. the code of a computational experiment, the source code file and the repository's revision number would be sufficient.

2.4.2 Distributed Version Control Systems

Unlike in centralized version control systems, each client hosts a full copy of the repository on its local machine. This also means that any of the clients can serve as backup, in case the server repository dies. This difference between centralized and decentralized version control systems is visualized in Figure 2.8.

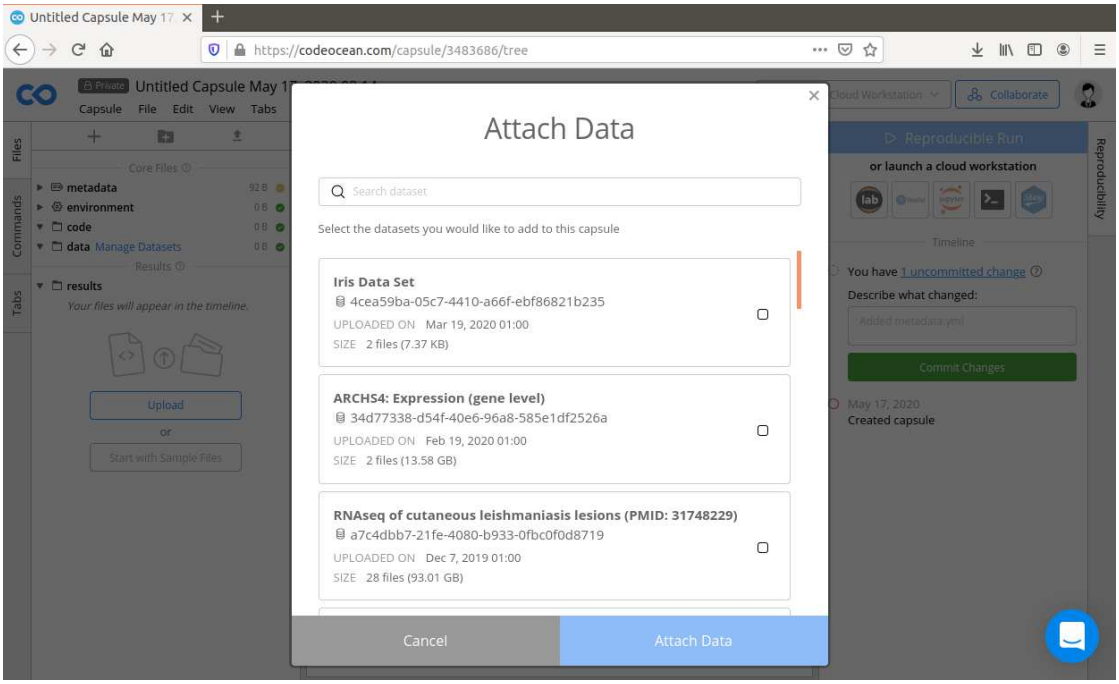


Figure 2.9: Screen Shot of the Dataset Integration Feature of Code Ocean

A central ideological difference to central version control systems is that distributed version control systems maintain change sets, whereas centralized ones maintain versions (i.e. a change of one file issues a new version of the central repository), therefore the distributed approach is more flexible when it comes to merging two branches [24].

Two prominent implementations are **Git**[35] and **Mercurial**[36], both initially released in 2005. Whereas Git is organized as a bundle of tools, Mercurial is one monolithic binary. There has been a long discussion on which tool is the better one [25], nevertheless with regard to the capabilities required for this thesis, they both meet the necessary requirements.

2.4.3 Code Ocean

Code Ocean is a research collaboration platform that can be used to publish source code[37]. It supports researchers in maintaining source code versions.

A core feature of Code Ocean is the ability to execute experiments via the browser directly on Code Ocean infrastructure. It also provides the possibility to include predefined datasets that can be used within the experiment as depicted in Figure 2.9.

[35] `https://git-scm.com/` (visited on: 10/13/2020)
[36] `https://www.mercurial-scm.org/` (visited on: 10/13/2020)
[37] `https://codeocean.com/` (visited on: 10/13/2020)

Although the tool already provides a lot of useful features in terms of experiment reproducibility, the tool lacks a proper research data management system. No datasets can be uploaded and it is not possible to include only subsets of the available datasets. Furthermore, the available datasets are not versioned, only the current version is available for the experiments.

2.5 Summary

This chapter gave an overview of recent developments in the field of research data management. For implementing a service that enables researchers to publish research datasets and computational experiments that persistently reference subsets of this datasets there are already relevant guidelines and recommendations.

The FAIR Guiding Principles discussed in Section 2.1 provide four principles to achieve reusability of published content (datasets, source code, etc.) which is one of the goals stated in Section 1.2.

In order to identify data subsets in computational experiments, a facility for persistently citing this subsets in a human and machine-readable way has to be provided. The RDA Recommendations for Data Citation already give suggestions on how to design a system that provides this citation capability.

Although persistent identifiers are part of the solution presented in this thesis, using a system that relies on persistent identifiers only is not sufficient. They provide an infrastructure to persistently reference published resources, nevertheless they cannot guarantee that the referenced resources stay unchanged (e.g. records in the dataset were updated).

Therefore, a research data management service that is able to track all modifications made to a dataset and identifies specific versions of dataset in order to cite them in a publication (human-readable form) or a computational experiment (machine-actionable). As discussed there is a wide variety of research data management solutions but none of them meets all the requirements, therefore a new RDM solution is needed that is capable of identifying subsets of dynamic datasets.

On the other hand, versioning of source code is already well-established and a wide variety of tools exist to manage source code and to identify specific versions of it. A more detailed discussion on which systems were used for this thesis is given in Section 3.1

CHAPTER 3

Conceptual Design

The main goal of this chapter is to describe how a research data repository can be implemented in a way that it meets the requirements described in Section 1.3. Figure 3.1 shows the draft of a desired solution for publishing computational experiments. Researchers can publish their data to a research data repository which keeps track of all changes. In addition, metadata describing the dataset is attached to it. Once a researcher wants to use the data in his experiment, he can query the repository where he retrieves a persistent identifier (PID) along with the resulting dataset. Subsequently, the data is referenced in the experiment using the PID. This way, exactly the same subset that was initially used for the experiment can be identified.

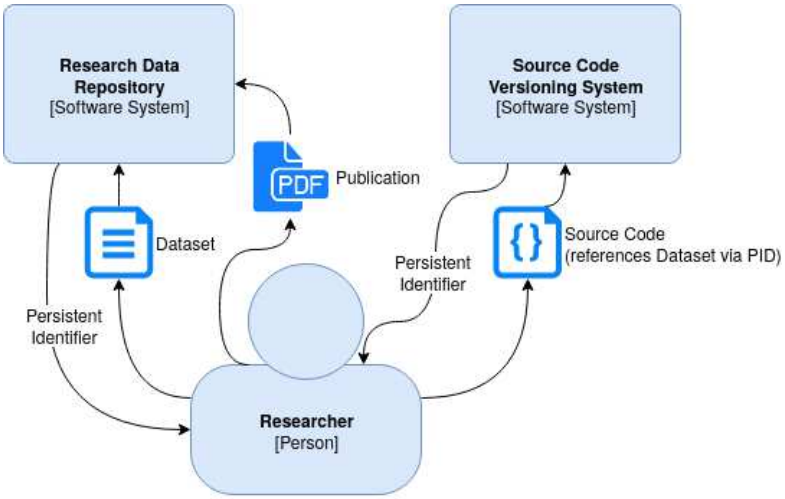


Figure 3.1: Desired scientific workflow

The source code of the experiment is versioned by a source code versioning (SCV) system as well. When publishing results of an experiment, a PID for the current version of the code is issued, which is attached to the paper along with the PIDs identifying all used datasets. Therefore, we do not only want to cite literature the publication relies on, but also the used source code and datasets.

This involves the development of a CKAN extension (discussion on why CKAN was chosen for this work can be found in Section 3.1.1) that offers the required data citation capabilities. A central requirement of the proposed solution is that it should be possible to migrate already existing data stores to the new solution. In addition, the solution should follow the RDA WGDC recommendations for data citation discussed in Section 2.2.

As there are already existing repository solutions, this chapter will start with a discussion on which repository solution will be used as a baseline. Subsequently, the architecture of the suggested system will be described using the C4 model, which will be explained in Section 3.2.

3.1 Discussion on Technology Choices

3.1.1 Repository Solution

In order to implement a repository solution that supports citation of data subsets, the augmentation of an existing product was favored. Due to the variety of research data repository products, it is not easy to decide which platform to choose. Therefore, a discussion on the advantages and disadvantages of the possible software products is of importance.

In 2017, Amorim et al. published a comprehensive comparison of state-of-the-art research data management platforms including the repository solutions presented in Section 2.3.1 [21]. Table 3.1 gives a comparison of the already presented repository solutions on feature level (based on the data provided by the paper of Amorim et al.). The software solutions only differ in minor details, e.g. the embargo feature which is not implemented for CKAN.

Nevertheless CKAN, was favored for this thesis. The main reason for this decision was the fact, that it already provides a rich datastore API that can be utilized to implement a desired datastore that complies with the Recommendations for Data Citation.

3.1.2 Data Store Technology

In order to generate subsets of uploaded datasets, we use a database management system. This section discusses whether a relational or NoSQL database should be used for storing the data records.

As the reference CKAN datastore implementation already uses a relational database (PostgreSQL), using the same technology seems self-evident. As already discussed by Säuerl [26], versioning an evolving schema can be a cumbersome task.

Feature	DSpace	CKAN	ePrints
Architecture			
Deployment	Installation package or service	Installation package	Installation package or service
Storage location	Local or remote	Local or remote	Local or remote
Maintenance costs	Infrastructure management	Infrastructure management	Infrastructure management
Open Source	✓	✓	✓
Customizeation	✓	✓	✓
Internationalization support	✓	✓	✓
Embargo	✓	Private Storage	✓
Content versioning	x	✓	✓
Pre-reserving DOI	✓	x	✓
Metadata and dissemination			
Exporting schemas	Any pre-loaded schemas	None	DC, METS, MODS, DIDL
Schema flexibility	Flexible	Flexible	Fixed
Validation	✓	x	✓
Versioning	x	✓	✓
OAI-PMH	✓	x	✓
Record license specification	✓	✓	✓

Table 3.1: Comparison of Research Data Repository solutions based on the publication of Amorim et al. [21]

By leveraging the flexibility of a schema-less NoSQL database, this thesis aims to point out a different way of implementing versioning of scientific datasets. Being the world's most widely used document store, MongoDB is the technology of choice[1].

As there is no need for an evolving data schema for the query store by the RDA recommendations the query store uses a relational database. As CKAN already uses a PostgreSQL database, the same technology was used for the query store.

3.1.3 Source Code Repository

The proposed solution also involves a source code versioning system. A central requirement to this component is its ability to identify specific versions of the experiment source code it maintains.

According to the Black Duck Open Hub, a public directory for free and open source software, most registered projects use Git as depicted in Figure 3.2. Therefore, a Git based solution is favored for versioning of the source code of the computation experiments.

To minimize dependencies to external services (e.g. GitHub) during the evaluation of the suggested approach, a local GitLab[2] server is used for this thesis.

[1] According to the DB-Engines ranking of document stores (`https://db-engines.com/en/ranking/document+store`) (visited on: 10/13/2020)

[2] `https://about.gitlab.com/` (visited on: 10/13/2020)

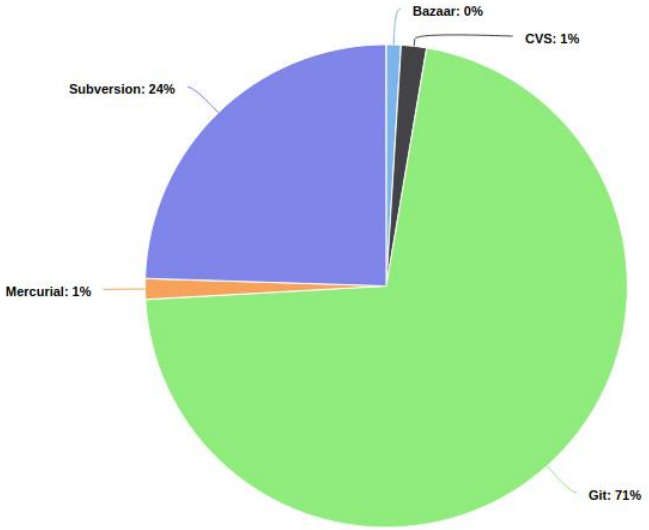


Figure 3.2: Popularity of source code versioning technologies.[3]

3.1.4 Persistent Identifier System

Although the DOI service is by far the most common solution (cf. Figure 2.2), the Handle PID service was used for the solution proposed in this thesis.

As Handle allows to setup a LHS that is independent from the GHR which can be used for the evaluation of the suggested approach and DOI is based on Handle from an infrastructural point of view Handle was favored for the solution proposed by this thesis.

PURL was not considered for this thesis, as it does not provide any advantages in comparison with the Handle service.

3.2 System Design

This section gives an overview of the architectural design of the system. For describing the architecture, the C4 Model introduced by Simon Brown is used[4]. According to the C4 Model, there are four levels of complexity by which a system can be described: system conext, container level, component level, and class level.

The **system context** describes how a system interacts with its environment. It shows the interaction between the system, its stakeholders and other external systems. The next level of detail is called the **container level**. It describes the interaction between a system's containers, which are defined as deployable pieces of software (e.g. web applications). The third level of detail, the **component level**, describes software pieces

[3] `https://www.openhub.net/repositories/compare` (visited on: 10/13/2020)
[4] `https://c4model.com/` (visited on: 10/13/2020)

that are related to each other in order to provide a certain feature. Such components are accessible via a defined interface. Class level, the fourth and most detailed level, will be covered in the next chapter.

3.2.1 System Context

A main task of the suggested solution to the problem described in Section 1.2 is publishing of datasets used in computational experiments that are published in code repositories. Two stakeholders interacting with the described system were identified: researchers and consumers.

The interactions between **reseachers** and the system can be described in two use cases. The first is publishing scientific results. This includes making all assets that were used to obtain the results (e.g. source code, datasets) available as well as the publication itself (e.g. in the form of pdf files). The second scenario is reproducing published results. In this case the researcher retrieves all assets in the exactly same state as they were used when the results were obtained in the first place to re-execute the experiment on his local machine.

Consumers of scientific results, on the other hand, are just interested in retrieving state-of-the art results (e.g. with possible corrections of mistakes in the datasets) in order to support their own claims or to conduct new experiments. An important aspect is the need for citeability of these assets.

3.2.2 Container Level

The container level diagram visualizes the relation between the containers of which the software is composed. The suggested solution consists of four independent applications, a CKAN instance, a CKAN datapusher service and a Source Code Repository and a Handle registry service.

Furthermore, there are four different storage containers. The CKAN instance requires a storage to store all uploaded files and a database to maintain application-specific data (e.g. credentials) and meta data of resources. These two containers are required by a default CKAN installation, the remaining two storage containers, data and query store, are specific to this solution. There are several suggested solutions for implementing a data store, but this thesis provides novel data store implementation by leveraging a NoSQL database. The query store stores all information required by the RDA Recommendations for Data Citation.

Once dataset files (e.g. csv files) are added to the CKAN instance as new resources, the datapusher is notified. This service extracts the data records from the files and pushes them into the CKAN datastore. After the dataset file has been transformed into a data store resource, it is possible to access the data via CKAN. The full diagram is given in Figure 3.4.

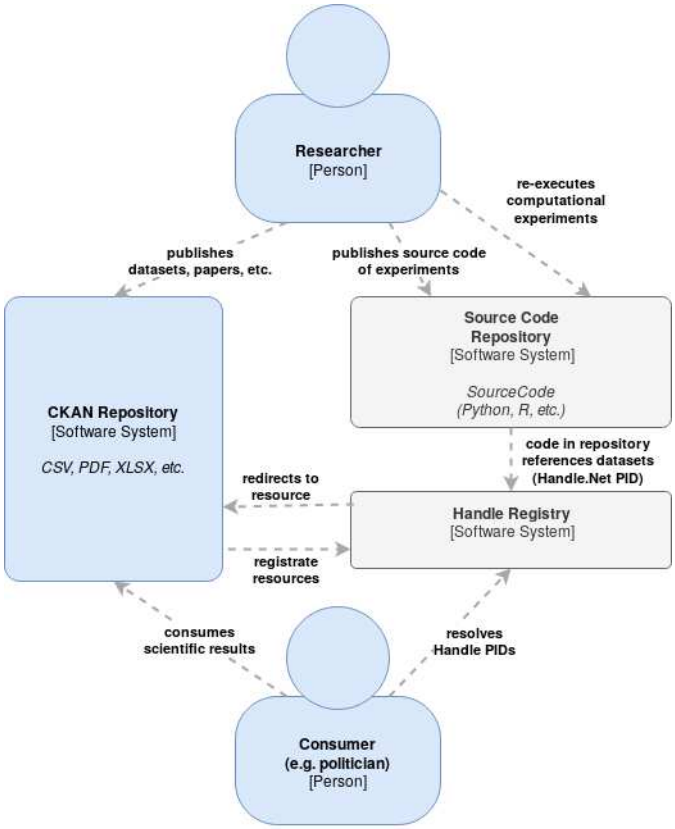


Figure 3.3: Conceptional Design: System Context

3.2.3 Component Level

The last level of detail, the component level, is depicted in Figure 3.5. As the main contribution of this thesis lies in the modification of a CKAN repository, the focus of this diagram lies on the CKAN extension. The grey components (CKAN and the CKAN DataStore API) are provided by the CKAN project. The UI components (yellow), the core components (red) and the storage access controller (green) were implemented within the scope of this thesis. The novel components are provided as CKAN extension.

The connection between CKAN repository and the CKAN extension is the DataStore API provided by CKAN, which is an interface describing several operations that can be performed on the data store. These operations include the creation and removal of data stores and CRUD operations on specific data stores. The component "MongoDB DataStore Implementation" represents the implementation of these operations. As there is a need for further operations that can be performed on the query store (e.g. retrieving a dataset via PID) an additional REST API is implemented in the component

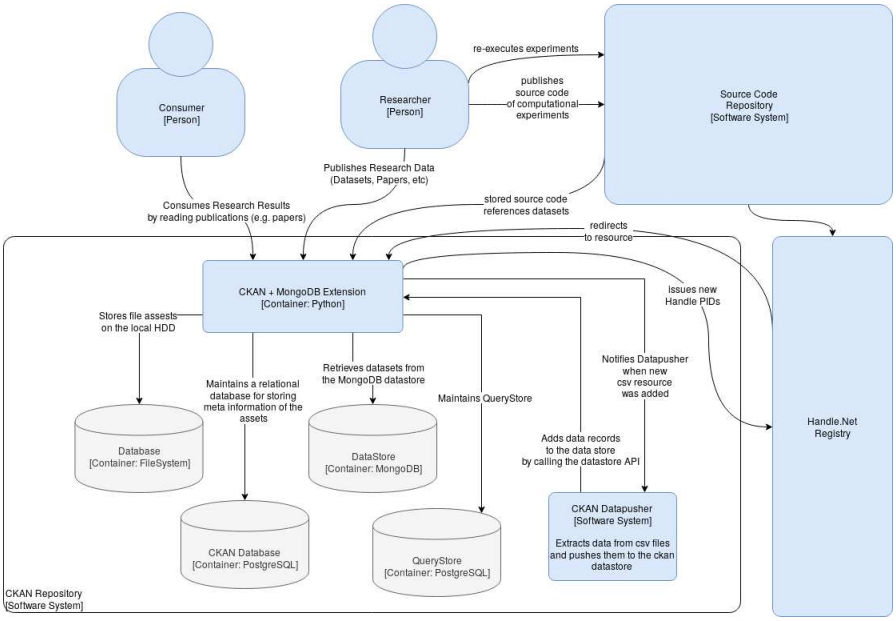


Figure 3.4: Conceptional Design: Container Level

"MongoDataStore API". Both components access the QueryStore and the MongoDB via the MongoDB Controller and the QueryStore Controller. These controllers can be seen as an abstraction layer for accessing the underlying databases.

To make use of the features in the user interface, there are Jinja templates for modifying the existing CKAN UI and associated controllers which are responsible for calling the according data store operations.

3.3 Summary

This chapter provided a conceptual outline of how to implement a system which is capable of publishing a computational experiment based on persistent datasets. This was achieved by modifying an already existing repository solution in a way that it aligns with the RDA Recommendations for Data Citation.

To provide a way of citing subsets of datasets that were published in the repository solution, two client implementations are described that access the data assets via the repository's Rest API.

The next chapter will present detailed insights on how these concepts were transformed into a running system.

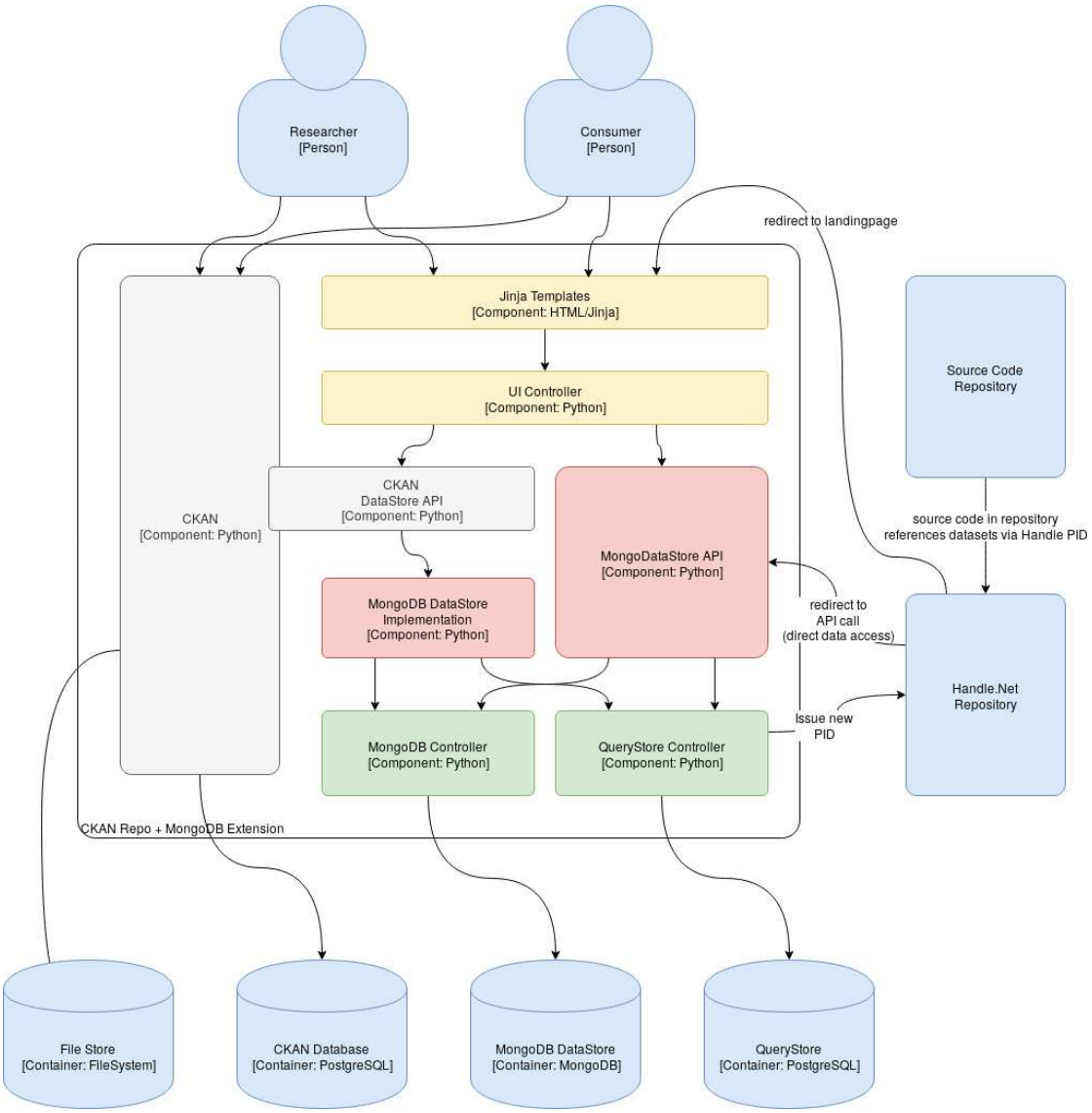


Figure 3.5: Conceptional Design: Component Level

CHAPTER 4

Implementation

This chapter discusses in detail how the proposed conceptional design from Chapter 3 was implemented. As for source code versioning systems and PID registries there are ready-to-use solutions that can be used for the presented solution, this chapter focuses on the implementation of a novel datastore for CKAN that provides the features described in the previous chapters. Besides that, a CLI tool is developed for retrieving data subsets that are identified by a Handle.Net PID which was issued by the DataStore implementation.

In order to enhance the functionality of CKAN, extensions have to be added to the repository. These extensions consist of one or more plugin packages that contain a class which implements one of the plugin interfaces provided by the repository. A complete description of the plugin interfaces can be found in the CKAN documentation.[1]

Table 4.1 gives an overview of all implemented software artifacts. The CKAN extension `ckanext-archiveview` was not included into the `ckanext-mongodatastore` plugin, as it serves a more general purpose (providing a resource view for zip archives). As the only Python package for accessing Handle.net services only supports Python versions below 3.6, a new Python package for accessing these services based on Python 3.8 was implemented.

The following sections will cover implementation details of the described assets.

[1] `https://docs.ckan.org/en/2.9/extensions/plugin-interfaces.html` (visited on: 10/13/2020)

Name	Description	DOI
ckanext-mongodatastore	CKAN extension implementing a new datastore backend and its related resource views.	10.5281/zenodo.4082139
ckanext-mongodatastore-evaluation	Code repository that contains all evaluation scripts as described in Chapter 5.	10.5281/zenodo.4082146
ckanext-archiveview	A CKAN resource view to display ZIP archives (for displaying tagged source code archives).	10.5281/zenodo.4082005
ckanext-mongodatastore-fetcher	A CLI tool to persistently retrieve data subsets from the CKAN repository	10.5281/zenodo.4082032
easyhandle	A Python package to access Handle.net naming services.	10.5281/zenodo.4082036

Table 4.1: Overview of implemented software artifacts that were required to build the system described in Chapter 3

4.1 Mongo Datastore Extension

This CKAN extension consists of three Plugins:

- `mongodatastore`
- `reclinecitationview`
- `landingpageview`

The `mongodatastore` plugin houses the datastore backend implementation, which interacts with the MongoDB instance and ensures that the datasets are stored in a versioned fashion. The `reclinecitaitonview` plugin is a slightly modified version of the `reclineview` plugin, which is shipped with CKAN. A detailed description of how we modified this resource view implementation in order to allow users to issue PIDs of data subsets is given in Section 4.1.10. The last plugin, `landingpageview`, is a resource view that displays the according landing page as described in Section 4.1.7 for URL resources of type `pid`, i.e resources containing a PID URL).

```
{
    "$or": [
        {"GDP1": {"$gt": 500}},
        {"Debt1": {"$lt": 200}}
    ]
}
```

Figure 4.1: A MongoDB query for retrieving all documents with a $GDP1$ greater than 500 or a $Debt1$ less than 200

4.1.1 Query Syntax and Normalization

In contrast to relational databases, MongoDB queries are represented as JSON documents. The aim of this section is to give a basic understanding of the query syntax. A full documentation of the query syntax can be found in the MongoDB documentation[2]. The simplest form of a query contains attributes with scalar values. Each attribute defines a filter that is applied on all documents, e.g. $\{'Country' : 'Austria'\}$ would return all documents in which the attribute $Country$ has the value $Austria$.

It is also possible to add logical conjunctions and comparative operations to the query. Figure 4.1 shows a query that retrieves all documents where the attribute $GDP1$ is greater than 1900 or $Debt1$ is less than 200.

In order to detect semantically equal queries, they have to be transformed to a normalized form. As for the queries, the order of their attributes has no impact on the query result, they are sorted alphabetically by key before they are processed by the database. As ordinary *Python* dictionaries do not preserve the order of attributes in versions lower than 3.6 [3], the data store uses the *OrderedDict* class for internal representation of queries.

4.1.2 Data Versioning and Timestamping

The first RDA Recommendation for Data Citation requires a versioning for all datasets maintained by the repository. This subsection will explain in detail how this was achieved by the proposed implementation.

As discussed in Section 3.1.2, MongoDB was used as database technology for storing datasets that are submitted to the repository. MongoDB is organized in collections that consist of several JSON objects which are referred to as documents. An example of this representation is given in Figure 4.2. Using the terminology of relational databases, the document corresponds to one row of a database table and each attribute represents a column and its associated value. Every document has an *_id* attribute by default,

[2] https://docs.mongodb.com/manual/tutorial/query-documents/#read-operations-query-argument (visited on: 10/13/2020)

[3] https://docs.python.org/3/whatsnew/3.6.html#whatsnew36-compactdict (visited on: 10/13/2020)

```
{
    "_id": "5de84c240ad6bd39eae14855",
    "_created": 2020-04-11T12:13:40.064+00:00,
    "Infl": 7.95483,
    "RGDP2": null,
    "Country": "Denmark",
    "Debt2": null,
    "RGDP1": 4539.63,
    "Debt1": 203.471,
    "Year": 1880,
    "GDP1": 840,
    "id": 1,
    "GDP2": null
}
```

Figure 4.2: Example of a data record represented as JSON document

which is of type *ObjectId* and maintained by the database. *ObjectId*s are 12-byte values consisting of a 4-byte timestamp 5-byte random value and a 3-byte incrementing counter.

In order to maintain older versions of a dataset, the data store must not overwrite data records once they are updated. Therefore, updates are added as new documents to the collection while the older version of the record remains unchanged. In other words, one document in the collection represents the state of a data record for a certain period of time. This requires another *id* in order to define the relation between data records and its states. In order to define which point of time a certain document was valid, two additional attributes, `_created` and `_valid_to`, are added to the documents. The first attribute indicates when the document was added to the collection and the second attribute indicates until when it was valid. Figure 4.3 gives an example of a deleted data record. In order to improve the performance on historic queries, a index is defined on the versioning fields (`_created` and `_valid_to`).

MongoDB provides an Aggregation Framework[4] that can be used to process documents in several stages, each stage performing an operation on the documents that are passed to this stage, e.g. operation *$match* only passes documents to the next stage that matches a condition.

The first approach was to use this aggregation framework in order to implement a versioning stage that filters all documents that are outdated, but according to the documentation each processing stage has a hard memory limit of 100MB[5]. This would

[4] `https://docs.mongodb.com/manual/aggregation/#aggregation-pipeline` (visited on: 10/13/2020)

[5] `https://docs.mongodb.com/manual/core/aggregation-pipeline-limits/`

```
{
  "_id": "5de84c240ad6bd39eae14855",
  "_created": 2020-04-04T12:13:40.064+00:00,
  "_valid_to": 2020-04-12T00:00:00.000+00:00
  "Infl": 7.95483,
  "RGDP2": null,
  "Country": "Denmark",
  "Debt2": null,
  "RGDP1": 4539.63,
  "Debt1": 203.471,
  "Year": 1880,
  "GDP1": 840,
  "id": 1,
  "GDP2": null
}
```

Figure 4.3: Example of a record that was deleted on the 4th of April 2020

```
{
      "_valid_to": {"$gt": <TIMESTAMP>},
      "_created": {"$lte": <TIMESTAMP>},
      ...
      <QUERY>
      ...
  ]
}
```

Figure 4.4: Filter frame which is used to query a resource at specific state

implicate that the solution is limit to datasets that do not exceed the size of 100MB (it would be possible to set a `allowDiskUsage` flag but this would implicate a significant performance impact).

Therefore, a query extension approach was chosen to restore a historic state of the dataset on querytime. The idea is that for each query conditions are added that only apply to the records that where valid at a certain point of interest. To achieve this the query is embedded in a query frame given in Figure 4.4, `<TIMESTAMP>` refers to the state of the resource upon which the query schould be executed and the placeholder `<QUERY>` represents the actual query.

Upon creation of a new DataStore resource information about the fields and their datatypes is passed to the DataStore implementation. Although MongoDB collections are not bound to a schema and therefore do not need to know the schema in advance,

```
{
    "_id":"5dea6f195535ea4666415d6e",
    "type":"text",
    "description": "English name of the country."
    "id":"Country"
}
```

Figure 4.5: Example of a field definition

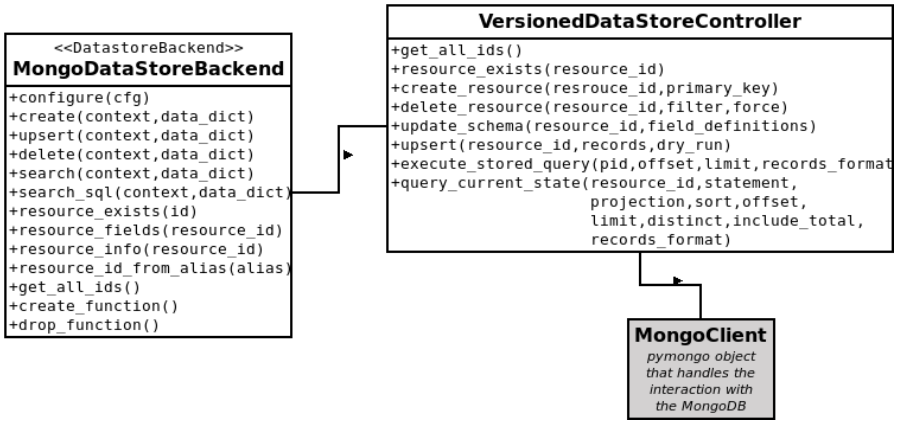


Figure 4.6: Class Diagram of the MongoDB DataStore backend implementation

this information is not discarded. It is stored in a separate collection that is named like the resource with an additional postfix *_fields*. An example of a column definition is given in Figure 4.5.

According to the API documentation of the CKAN datastore following datatypes are supported[6]: *text*, *json*, *date*, *time*, *timestamp*, *int* and *float*.

4.1.3 Implementation of the CKAN *DatastoreBackend*

Since CKAN version 2.7 it is possible to use a custom data store backend implementation. This is done by extending the *IDatastoreBackend* interface. In order to implement this interface a *registrate_backend* has to be added which maps a database technology to a custom datastore implementation.

Figure 4.6 gives a overview of the classes that are involved in realizing a custom datastore backend solution.

The **MongoDataStoreBackend** extends the DatastoreBackend implementation which is provided by CKAN. Its main concern is to validate input parameters and pass them on

[6] `https://docs.ckan.org/en/latest/maintaining/datastore.html#field-types` (visited on: 10/13/2020)

```
{
   "$or": [
      {"Infl": "Austria"},
      {"RGDP2": "Austria"},
      {"Country": "Austria"},
      {"Debt2": "Austria"},
      {"RGDP1": "Austria"},
      {"Debt1": "Austria"},
      {"Year": "Austria"},
      {"GDP1": "Austria"},
      {"id": "Austria"},
      {"GDP2": "Austria"}
   ]
}
```

Figure 4.7: Result of the query transformation for the term "Austria" based on the schema of the example presented in Figure 4.2

to the VersionedDataStoreController. For the *search* operations this class also performs a pre-processing of the query expression.

For this operation two ways of defining conditions are specified:

- **q**: with this parameter a fulltext search can be performed on the datastore. In the context of this thesis this means that a record is included in the result set if any of its attributes matches the passed term.
- **filter**: for this parameter a JSON object is expected, which specifies conditions for specific attributes. This representation already complies with the MongoDB query syntax.

In order to perform a fulltext search the query parameter (q) has to be transformed into a valid MongoDB query statement. An example of this transformation is given in Figure 4.7.

A full documentation of all methods defined by the DatastoreBackend base implementation can be found in the CKAN documentation[7].

The **VersionedDataStoreController** encapsulates access to the MongoDB in a way that all changes (inserts, updates and deletions) to the dataset are tracked. In order to achieve this, this access layer implements the versioning strategy presented in Section 4.1.2.

[7] `https://docs.ckan.org/en/latest/maintaining/datastore.html#ckanext.datastore.backend.DatastoreBackend` (visited on: 10/13/2020)

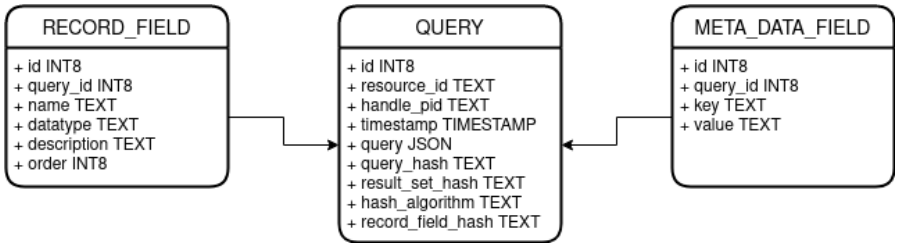


Figure 4.8: The Query Store entity model

4.1.4 Query Store

In order to re-execute queries that were submitted to the data store in the past, a facility is required that stores the submitted queries including all information necessary to re-execute the query in a way that it retrieves exactly the same dataset.

To meet this requirement, all queries including their metadata are stored into a relational database. The entity model (depicted in Figure 4.8) consists of three tables. The `QUERY` table contains all data that is required to re-execute a historic query and subsequently validate the result set and the `RECORD_FIELD` table contains descriptive information about the record fields. As different scientific communities may require different information for citing datasets, this information is stored as key-value entries in a separate table (i.e. *META_DATA_FIELD*). A detailed description of how these entries are assembled to a full citation text is discussed in Section 4.1.8.

The query entity also contains hash values for the query, result set and assigned record fields to identify query duplicates. Only in case a query differs in `query_hash`, `result_set_hash`, `resource_id` and `record_field_hash`, a new query entry is issued which is registered to the Handle service. A detailed description of the Handle.Net integration is given in Section 4.1.5.

4.1.5 Handle.Net Integration

For integrating a Handle.Net into the solution two approaches were considered:

1. Store all query metadata into the Handle.Net entry instead of a local database
2. Store query metadata into a local query store database and then issue a Handle PID that references the landing page of the query

Although the first approach seems to be less effort, the second aproach was chosen for this thesis. The reason for this choice is that when storing the query metadata into a local database the solution is less dependent on a specific PID technology. In this approach it is also less effort to modify the solution in order to support a different PID service (e.g. DOIs). All that is needed is to change the method that issues the PID.

```
{
  "responseCode": 1,
  "handle": "99.9999/14c30c2b-fb34-4c7d-9c1e-9d90515e309f",
  "values": [
    {
      "index": 1,
      "type": "URL",
      "data": {
        "format": "string",
        "value": "http://192.168.178.22:5000/querystore/view_query?id=212"
      },
      "ttl": 86400,
      "timestamp": "2019-12-09T08:53:40Z"
    },
    {
      "index": 2,
      "type": "STREAM_URL",
      "data": {
        "format": "string",
        "value": "http://192.168.178.22:5000/datadump/querystore_resolve/212"
      },
      "ttl": 86400,
      "timestamp": "2019-12-09T08:53:40Z"
    },
    {
      "index": 3,
      "type": "API_URL",
      "data": {
        "format": "string",
        "value": "http://192.168.178.22:5000/api/3/action/querystore_resolve?pid=212"
      },
      "ttl": 86400,
      "timestamp": "2019-12-09T08:53:40Z"
    },
    {
      "index": 100,
      "type": "HS_ADMIN",
      "data": {
        "format": "admin",
        "value": {
          "handle": "0.NA/99.9999",
          "index": 200,
          "permissions": "011111110011"
        }
      },
      "ttl": 86400,
      "timestamp": "2019-12-09T08:53:40Z"
    }
  ]
}
```


Figure 4.9: Example of an Handle.NET entry

In Figure 4.9 an exemplary Handle.Net entry is given. It contains four values of type *URL*, *STREAM_URL*, *API_URL* and *HS_ADMIN*. The *URL* refers to the human-readable landing page of the query, whereas the *API_URL* value points at the Rest API endpoint of the same landing page. The *STREAM_URL* contains a download link of the cited asset.

The Handle.Net PID can be resolved by visiting the web interface of the local registry instance. By providing a PID the user is forwarded to the location specified in the *URL* value of the Handle entry.

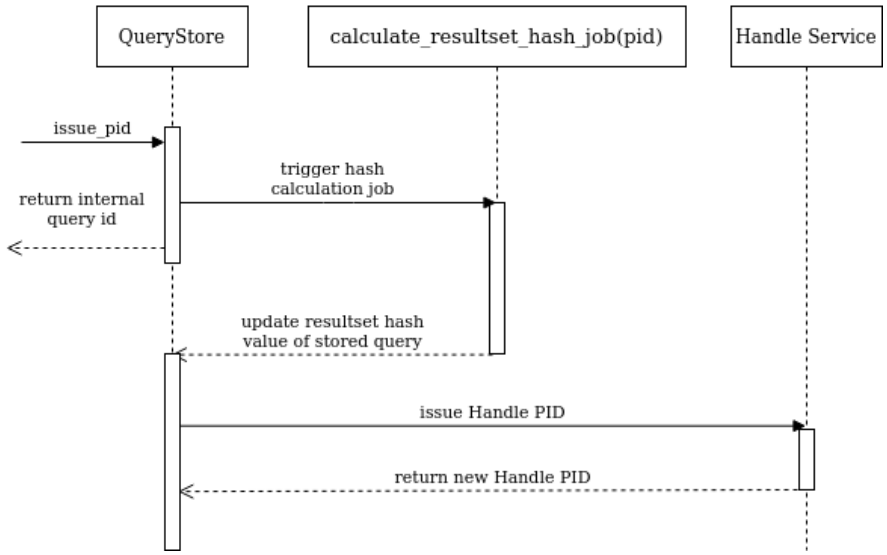


Figure 4.10: Sequence Diagram of the result set hash calculation procedure

4.1.6 Asynchronous Resultset Hash Calculation

As required by the RDA Recommendations for Data Citation (R6), each stored query should also include a hash value of its result set in order to be able to find duplicate queries (i.e. queries that have an identical query and result set hash value). In case of large datasets the calculation of the result set hash value can be a time-intensive task as it requires to iterate over each record.

Therefore, this task was implemented as asynchronous task that is triggered once a query is stored to the query store. This means that the result set hash is not calculated immediately. As a consequence also the Handle PID cannot be issued immediatly as for this step the result set hash has to be known.

Once the calculation is done, the job stores the result set hash to the query and assigns a persistent identifier to the query. In case of a duplicate query, the already existing identifier is re-used otherwise a new identifier is issued. Figure 4.10 gives an overview of the described result set hash calculation procedure.

CKAN already provides an interface for enqueuing background jobs[8]. The job itself is defined as a Python function that is passed on to the background job API.

4.1.7 Landing Page

The CKAN Datastore extension provides two machine-actionable and one human-readable way to re-execute stored queries.

[8] https://docs.ckan.org/en/2.8/maintaining/background-tasks.html

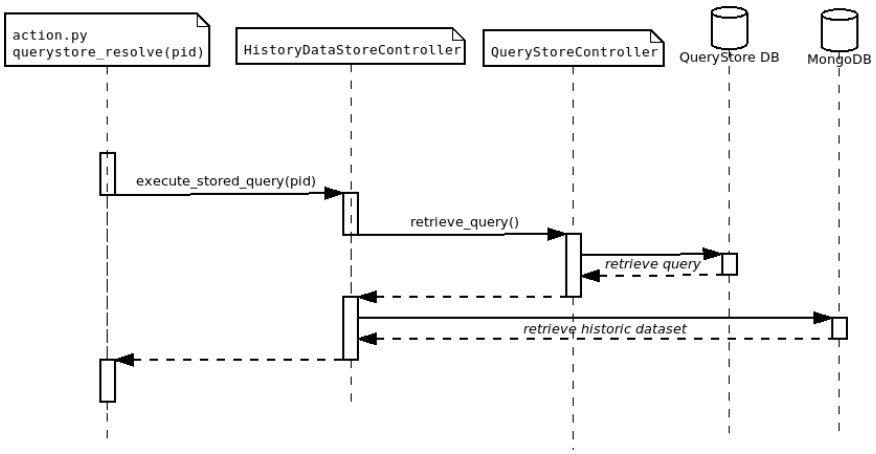


Figure 4.11: Sequence Diagram of the *querystore_resolve* operation

For a machine-actionable access to historic query results including all its metadata, a REST endpoint is provided. This is accomplished by registering a new action, `querystore_resolve`, which can be invoked via the CKAN Rest API:

```
[CKAN Host]/api/3/action/querystore_resolve
```

The endpoint expects an internal query id as input parameter. Optionally, an *offset* and a *limit* value can be provided (this is needed for paging through larger datasets). This action calls the *VersionedDataStoreController* which first retrieves the query store entry for the provided PID and subsequently performs the query on the dataset's state at the time when the query was executed the first time (for a better understanding this course of action is depicted in Figure 4.11).

Furthermore, there is another endpoint which does stream the data subset to the client without any metadata information. This endpoint can be used to download larger query results directly to the local file system. To directly stream the result of a stored query, following endpoint has to be called:

```
[CKAN Host]/datadump/querystore_resolve/[INTERNAL QUERY ID])
```

Besides the machine actionable endpoint, there is also a human readable way to access the same information. A new PID search box was added to the repository's main page which redirects to the according query landing page (Figure 4.12). This page consists of a UI controller (*QueryStoreUIController*) and a Jinja template that defines how the HTML response is rendered. By calling the *querystore_resolve* action and passing this information to the template the JSON response is rendered in a human readable way. The search box accepts either the internal query id or a Handle PID that was assigned to a stored query.

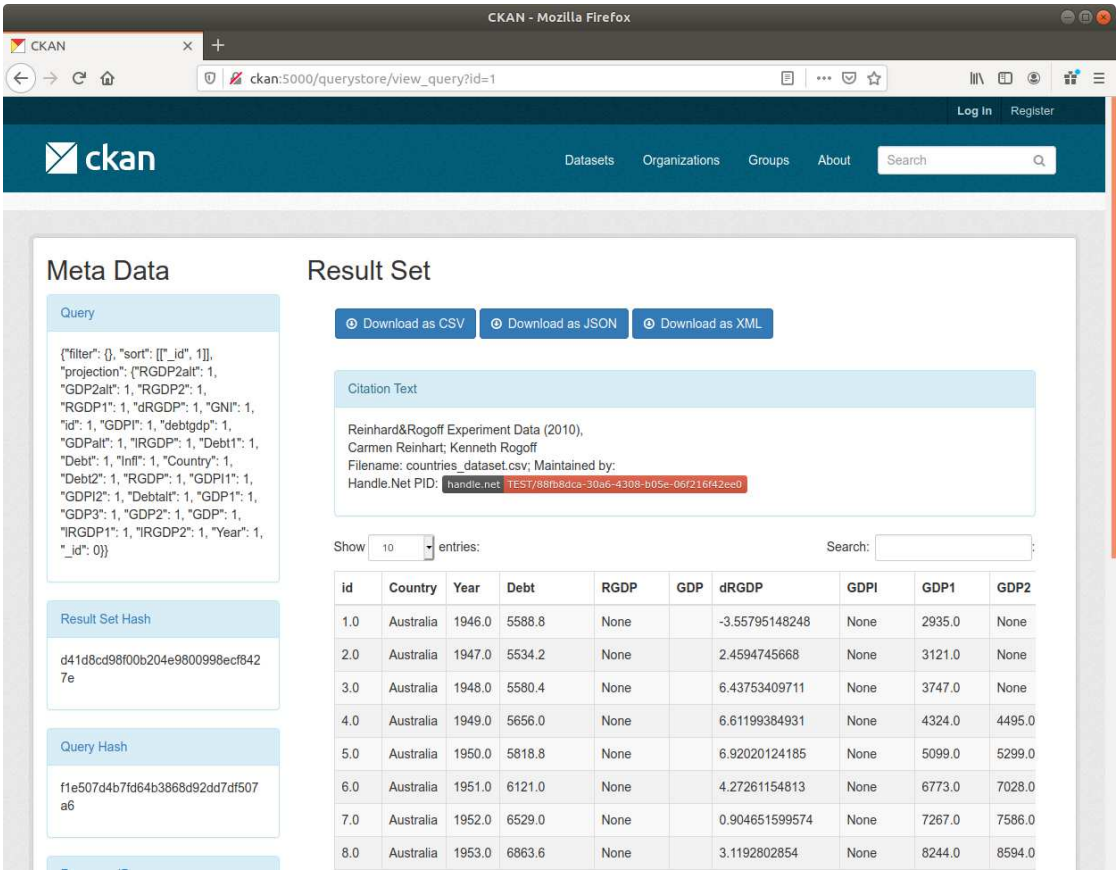


Figure 4.12: PID landing page

4.1.8 Citation Text Templates

Even in citation of literature there is no standard structure for citation texts. Depending on the domain, there are different citation styles that are accepted by the community, e.g. MLA Citation Style for the humanities or the Electrical Engineering Citation Style for natural sciences.

In order to provide a flexible solution, the view plugin was designed in a citation style agnostic way. This means that only the metadata key value pairs are retrieved from the backend (e.g. author name, year of publication, maintainer) and in the frontend a template defines the structure of the resulting citation text.

A template is represented as HTML snippet, where `<span/>` tags define placeholders for metadata fields. Once a query execution succeeds, every metadata field value is placed into the according tag identified by its `id` attribute. For a replacement of the current citation style, only the citation snippet in the templates snippet folder has to be replaced.

Figure 4.13 gives an example of a citation text template. The final citation snippet is

```
<p>
  <span id="citation_title"/>(<span id="citation_year"/>),
  <span id="citation_author"/><br/>
  Filename: <span id="citation_filename"/>;
  Maintained by: <span id="citation_maintainer"/><br/>
  Handle.Net PID: <span id="citation_handle_pid"/>
</p>
```

(a) Template

(b) Result

Figure 4.13: A simple example of an auto-generated citation text

displayed on the landing page, from where researchers can copy it into their publication.

4.1.9 Query Store Verification Tool

The CKAN Datastore Plugin also includes a commandline tool that re-executes all queries stored in the Query Store and verifies if the result set hash values are still valid. This tool can be used after a technology migration (e.g. updating the version of MongoDB) to verify if all stored subsets are still persistently identifiable.

To run the tool, the current working directory has to be changed to the installation directory of the plugin. Furthermore, the virtual environment in which CKAN is running has to be activated. After these preconditions are met, calling `ckan -c [PATH TO CKAN CONFIG] mongodatastore_check_integrity` invokes the integrity check.

4.1.10 Recline Citation View

Due to the fact that the MongoDB Datastore plugin discussed in Section 4.1 aligns with the DataStore API, any view that uses this interface is able to interact with this implementation. Nevertheless, the user would not be able to issue a persistent identifier of the generated subsets. Therefore, a proper solution also requires a resource view that enables the user to issue a persistent identifier of the currently queried subset of the data resource.

```
__type__: 'name-of-backend' // e.g. elasticsearch

// Initial load of dataset including initial set of records
fetch: function(dataset)

// Query the backend for records returning them in bulk.
// This method will be used by the Dataset.query method to search the backend
// for records, retrieving the results in bulk.
query: function(queryObj, dataset)

// Save changes to the backend
save: function(changes, dataset)
```

Figure 4.14: recline.js Backend API

Recline Architecture

CKAN already offers the ReclineView plugin, which is capable of displaying tabular data resources. This plugin uses *recline.js*[9] for displaying the data. This JavaScript framework consists of three components: Backend, Model and View.

The **Backend** is an abstraction layer that encapsulates access to a remote datasource (e.g. a Google Docs Spreadsheet, Solr, elasticsearch). Each datasource needs a custom implementation. In this case the ReclineView uses a CKAN backend implementation[10]. The API specification of recline.js backends is given in Figure 4.14.

Models are structures maintaining the data which is subsequently presented in the view. The Model class used in this plugin is the *Dataset*. It is a collection of records and also contains all information concerning which backend implementation to use.

Views are Web User Interfaces components that are responsible for displaying Model objects within a web page.

Implemented Modifications

For being able to issue PIDs for the current query a button and a panel `div` were added to the recline_view.html file as listed in Figure 4.15. Pressing the `Cite Subset` button invokes a method that accesses the current query state and passes it to the CKAN Datastorebackend via the `issue_pid` endpoint. As a result, the view receives the internal id of the stored query. This id is used to show a link to the stored query's landing page. Once the calculation of the result set hash and the assignment of a Handle PID is finished, the user can retrieve the Handle PID form this landing page.

[9] `https://okfnlabs.org/recline/` (visited on: 10/13/2020)
[10] `https://github.com/okfn/ckan.js` (visited on: 10/13/2020)

```
...
<div data-module="recline_view"
  data-module-site_url="{{ h.dump_json(h.url('/', locale='default', qualified
      =true)) }}"
  data-module-resource="{{ h.dump_json(resource_json) }}" ;
  data-module-resource-view="{{ h.dump_json(resource_view_json) }}" ;
  data-module-map_config="{{ h.dump_json(map_config) }}" ;
  data-module-dataproxy-url="{{ h.get_dataproxy_url() }}">

  <a id="cite-btn" class="btn btn-default">Cite Subset</a>
  <div id="cite-response-panel" class="panel panel-info">
    <div class="panel-heading">
    Information
    </div>
    <div id="cite-response-text" class="panel-body">
    </div>
  </div>

  <div id="datagrid">
    <h4 class="loading-dialog">
    <div class="loading-spinner"></div>
    <div class="left">{{ _('Loading...') }}</div>
    </h4>
  </div>
</div>
...
```

Figure 4.15: Modified recline_view.html file

4.2 Archive View

As the archive view provides a more general purpose, we decided to put this feature into a separate CKAN extension.

The extension parses zip files that were uploaded to the repository or were referenced via URL and presents the archives file structure using the jsTree framework.

4.2.1 Zip Archive Parsing

For rendering the internal structure of the uploaded zip archive resources, this structure has to be transformed into Python specific representation. For this extension, we chose dictionaries to reflect the relation between files and directories. The keys of the dictionary represent file or directory names. The value defines if the item is a file or directory. For files the value is set to `None` and for directories the value is set to a new dictionary which represents the structure of the subdirectory. Figure 4.16 demonstrates how to plugin transforms zip archives into Python dictionaries.

```
florian@ffw:~/git/ckanext-mongodatastore-evaluation/data/datasets$ tree filter_values
filter_values
├── int_client_port
├── int_http_response_header_length
├── int_server_port
├── str_client_headers
├── str_client_ip
├── str_modified_expires_headers
└── str_server_ip

0 directories, 7 files
```

(a) Internal Structure of the example archive.

```
1  {
2    'filter_values': {
3      'int_client_filter': None,
4      'int_http_response_header_length': None,
5      'int_server_port': None,
6      'str_client_headers': None,
7      'str_client_ip': None,
8      'str_modified_expires_headers': None,
9      'str_server_ip': None,
10   }
11 }
```

(b) The Python representation of the zip archives internal structure.

Figure 4.16: A example of how the archive view extension converts the internal structure of a zip archive into a Python dictionary.

4.2.2 File Tree View

For recursively rendering the internal structure of the according zip archive, the recursive loop feature of the Jinja templates is used. The backend provides the file structure represented as Python dictionary and the template transforms this structure into a unordered HTML list as depicted in Figure 4.17. Subsequently the JavaScript framework `jsTree` transforms this HTML code into a visually more appealing file tree view as depicted in Figure 4.18.

4.3 Client Implementations

To enable researchers to use the persistently identified subsets of the published datasets a component is needed that enables computational experiments to retrieve the data resources by identifying them via PID.

This thesis discusses two approaches to integrate datasets into computational experiments:

- Downloading the resource to the local files system which is subsequently accessed throughout the execution of the experiment (by providing a CLI tool)

```
<ul>
   {% for key, value in file_tree_dict recursive %}
      {% if value %}
         <li data-jstree='{"opened": "True", "icon":"jstree-folder"}'>{{ key
             }}
            <ul>{{ loop(value.items()) }}</ul>
         </li>
      {% else%}
         <li data-jstree='{"icon":"jstree-file"}'>
            {{ key }}
         </li>
      {% endif %}
   {% endfor %}
</ul>
```

Figure 4.17: Algorithm for transforming the Python file structure representation into a HTML unordered list.

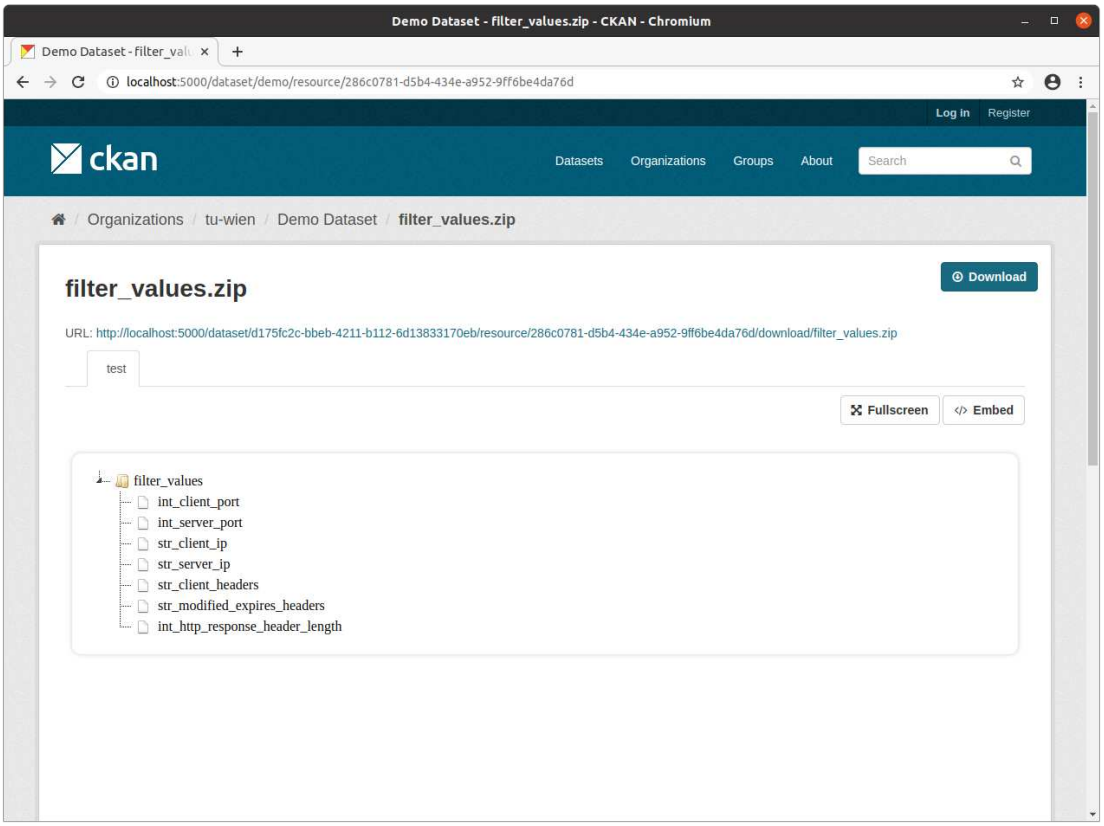


Figure 4.18: Screenshot of the archive view.

```
ckanfetch -f csv \
       -t ./data.csv \
       --include-header \
       --cache \
       TEST/96d91504-9673-46d8-9f55-02bc001d841d
```

Figure 4.19: Example usage of the `ckandf` tool

- Provide a software module that allows to reference the data within the experiment execution, i.e. the data is not written to the file system.

A separate solution for each case is provided to allow a versatile usage of the proposed system. Whereas the first approach is idependent from the programming language that is used in the experiments it requires access to the local disk and enough storage capacity. On the other hand the second option is bound to a certain language and is limited to the available RAM capacity but has the advantage of being able to page through the dataset, e.g. retrieve the result set in chunks of 100 records.

In the scope of this thesis a Python tool covering the first and a R package covering the second approach are provided. Both solutions are capable of retrieving a data subset identified by a PID that was generated from the proposed CKAN datastore implementation.

4.3.1 `ckanfetch` Command Line Tool

This component is a command line tool that is implemented in Python and published via the `pip` package management system. In case an installation via package manager is not possible the source code of the CLI tool is available on GitHub[11].

The tool retrieves the according Handle entry of the provided PID and uses the URL provided in the field `STREAM_URL` to streams the resource to a specified location. An example of how to use the tool is given in Figure 4.19.

4.4 Summary

This chapter gave insight into the implementation details of the proposed solution.

In order to provide a running prototype of the concept proposed in the previous chapter, two CKAN extensions consisting of several Plugins were implemented. The central part of the presented solution is the datastore implementation. The novelty of this datastore implementation is its usage of a NoSQL database for storing data records in a versioned

[11] `https://doi.org/10.5281/zenodo.4082032` (visited on: 10/13/2020)

way and it also has an incorporated query store, that keeps track of all queries that are submitted to the datastore.

Besides the datastore, several views were implemented for displaying landing pages (`landingpageview`), issuing PIDs for data subsets (`reclinecitationview`) and displaying zip archives (`archiveview`).

In addition, we implemented a Python CLI tool that is capable of retrieving historic datasubsets identified by a Handle PID.

As there was no suitable Handle.net client implementation, we also implemented a package (i.e. *easyhandle*) to handle the communication between CKAN and the Handle.Net service.

CHAPTER 5

Evaluation

The goal of this chapter is to provide a comprehensive evaluation of the solution presented in this thesis. This chapter is organized in five sections, starting with a description of the test environment in Section 5.1. This covers a specification of the host machine on which the experiments were conducted, instructions on how to set up the required services and how to re-execute the evaluation experiments. Sections 5.2 and 5.3 examine to what extent the provided solution meets the described functional and non-functional requirements of Section 1.3. The remaining Sections 5.5 and 5.6 discuss to what extent the presented solution aligns with the *RDA Recommendations for Data Citation* and *FAIR Principles*.

5.1 Test Environment

5.1.1 Hardware Specification

In order to make the test results comparable, all evaluation experiments are executed on the same host. Table 5.1 lists the specification of the machine used.

5.1.2 Evaluation Software

All evaluation experiments described in this chapter are provided in a separate code repository on GitHub[1]. This project covers two concerns:

- providing docker-compose files for an easy setup of the proposed solution
- providing a python script to execute all described experiments

[1] `https://doi.org/10.5281/zenodo.4082146` (visited on: 10/13/2020)

Hardware	
CPU	AMD Ryzen 5 1500x quad-core processor x 8
RAM	32 GB
HDD	Samsung 850 EVO interne SSD 250GB

Software	
Operating System	Ubuntu 20.04 LTS
Python	3.8
CKAN	2.9

Table 5.1: Hardware and Software specifications of the Test environment

Service	Exposed Port
ckan (with mongodatastore plugin)	5000
datapusher	8800
postgresql	5432
solr	8983
redis	-
handle.net	2641, 8000
gitlab	8081
mongodb config server	-
mongodb shard01	-
mongodb shard02	-
mongodb shard03	-
mongodb router	27017

Table 5.2: Services included in the default docker-compose file

Docker Compose Files

All docker-compose files are located in the subfolder `docker-setup` of the repository. It contains one default configuration (i.e. `docker-compose.yml`) which sets up the default system configuration as described in Table 5.2. As the performance of this configuration is compared against the default datastore plugin provided by CKAN, another docker-compose file (`docker-compose.legacy.yml`) is provided which starts up a CKAN instance with the default datastore plugin installed and all depended services.

Besides evaluating the performance of the proposed solution, this chapter also discusses how to improve the achieved results. Database sharding is one concept that is considered in the course of this discussion.

For document based database systems like MongoDB, sharding is a common way to handle higher loads in productive environments. Figure 5.1 shows an exemplary database setup where the stored data is split up into two shards. There are three types of instances involved: shards, router and config servers. Shards are database instances that host

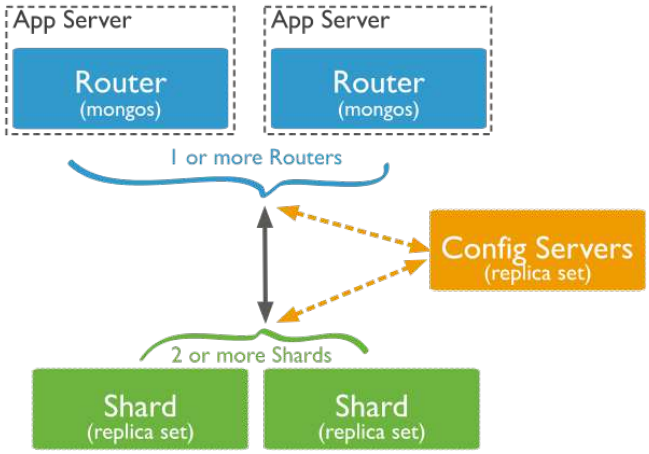


Figure 5.1: Components of a sharded MongoDB database

subsets of the collections, router instances are the interface between applications (in this case the CKAN repository) and the shard instances and config servers are responsible for storing metadata of the cluster.

As increasing the number of shards is one way to increase the performance of an application that is based on a MongoDB, Section 5.3.6 discusses the performance impact of horizontal scaling on MongoDB databases in more detail.

In order to investigate how the number of shards affects the performance of the system, several docker-compose files are provided with different MongoDB configurations. The files are located in the `docker-setup` folder and follow the naming schema `docker-compose.shardXX.yml` where XX represents the number of shards used.

Test Execution

The test case implementations of all functional and non-functional tests are located in following folders:

- `evaluation/tests/functional`
- `evaluation/tests/nonfunctional`

The steps required for setting up the evaluation tool are covered by the `README` page of the evaluation project. The `run_testcases.py` script is used to execute the evaluation experiments. It is possible to define a tag which describes the current test run configuration. This tag is included into the name of the result files, which makes it easier to distinguish between results of the different test runs. Furthermore, it is possible to specify which experiment to execute by providing `--functional` or `--nonfunctional` parameters.

Valid parameters are (according to the test cases defined in Section 5.2 and 5.3):

- `--functional`:
  - 1.1
  - 2.1
  - 2.2
  - 2.3
  - 3.1
  - 3.2
  - 3.3
  - 3.4
  - 4.1
  - 4.2
  - 4.3
  - 4.4
  - 5.1
  - 5.2
- `--nonfunctional`:
  - 1
  - 2
  - 3
  - 4

For the evaluation of this thesis, every single execution of this script on a specific docker-compose configuration is denoted as test run. The `results` folder contains a separate sub folder for each test run which contains csv files (e.g. response times), log files (e.g. log files from executed R scripts of functional test 5.1 and 5.2) and pdf files (e.g. results from the executed R scripts of functional test 5.1 and 5.2).

To generate all evaluation diagrams presented in this thesis, the csv result files have to be copied to the `charts/data` directory and the `generate_diagrams.py` script has to be executed.

5.2 Evaluation of Functional Requirements

This section verifies if the proposed systems provides the required functionalities described in Section 1.3. An overview of all defined use cases is given in Table 5.3. By transforming the different use cases into functional test cases, it is possible to evaluate to what extent the provided solution corresponds to the use cases. Besides a description of which steps are executed within the test the test case also defines a set of preconditions and postconditions. Table 5.4 gives an overview of all test cases and how they relate to the use cases.

5.2.1 Test Assets

All functional tests are based on the experiment source code provided by the publication of Herndon, Ash, and Pollin which was published on the website of the University of Massachusetts Amherst [2]. As not all files in the provided archive were needed and some modifications had to be performed on the original files we assigned a DOI to a deposit that contains all original files[3] and another DOI to a deposit the consists of all modfied

[2] `https://www.peri.umass.edu/images/WP322HAP-RR-GITD-code-2013-05-17.zip` (visited on: 10/13/2020)

[3] `https://doi.org/10.5281/zenodo.4017423` (visited on: 10/13/2020)

Identifier	Description
UC 1	publish datasets
UC 1.1	maintain metadata of datasets
UC 1.2	upload resources to datasets
UC 2	update datasets
UC 3	query resources
UC 3.1	define filter on specific fields
UC 3.2	define sort order of the result set
UC 3.3	issue PIDs for queries
UC 4	retrieve persistently identified published experiments
UC 4.1	retrieve persistently identified source code
UC 4.2	retrieve persistently identified datasets

Table 5.3: Overview of all Functional Test Cases

files[4] that are effectively used by the test cases. The following Section 5.2.2 describes in detail the required modifications to the original files.

5.2.2 Test Asset Preprocessing

RR_preprocessed.dta To this file two modifications were applied:

1. The file was transformed to a csv file using RStudio.
2. A new column 'id' was introduced and for each record an incrementing id value was set.

The resulting file is named `countries_dataset.csv`

RR.R This file contains the code of the baseline experiment used for the functional test cases 5.1 and 5.2. In the original code a subset of the provided dataset is used for the conducted experiment. Listing 5.2 shows the according code snipped.

As in the functional tests this subset is retrieved from the CKAN repository only the subsequent experiment code is relevant for the functional test cases.

Out of the remaining code two different versions of the experiment were created:

1. An experiment that expects a file containing a PID in the same directory, which is used to resolve the actual API call to the repository, that persistently returns the data subset.

[4] `https://doi.org/10.5281/zenodo.4082020` (visited on: 10/13/2020)

Identifier	Related Use Cases	Tested Functionality
FTC 1.1	UC 1, UC 1.1, UC 1.2	Upload of a CSV file to the repository along with its metadata
FTC 2.1	UC 2	Inserting new records to an CKAN resource
FTC 2.2	UC 2	Modification of a record in an CKAN resource
FTC 2.3	UC 2	Deletion of a record in an CKAN resource
FTC 3.1	UC 3.1	Querying the current state of an CKAN resource by exact value
FTC 3.2	UC 3.1	Querchying the current state of an CKAN resource by defining a range on a specific field (range query)
FTC 3.3	UC 3.1	Fulltext Query on an CKAN resource
FTC 3.4	UC 3.2	Query with defined sort order on an CKAN resource
FTC 4.1	UC 3.3, UC 6.2	Creation and retrieval of an persistently identified subset of an CKAN resource (defined by an exact filter condition)
FTC 4.2	UC 3.3, UC 6.2	Creation and retrieval of an persistently identified subset of an CKAN resource (defined by range query)
FTC 4.3	UC 3.3, UC 6.2	Creation and retrieval of an persistently identified subset of an CKAN resource (defined by fulltext query)
FTC 4.4	UC 3.3, UC 6.2	Creation and retrieval of an persistently identified subset of an CKAN resource (defined query with sort order)
FTC 5.1	UC 3.3, UC 4, UC 5, UC 6, UC 6.1, UC 6.2, UC 6.3	Persistently identify an computational experiment and the data subset (using the REST API) it is based on + Publishing of both PIDs in CKAN along with related metadata
FTC 5.2	UC 3.3, UC 4, UC 5, UC 6, UC 6.1, UC 6.2, UC 6.3	Persistently identify an computational experiment and the data subset (using the CLI tool) it is based on + Publishing of both PIDs in CKAN along with related metadata

Table 5.4: Overview of all Functional Test Cases

```
## Cut to postwar analysis
RR <- subset(RR,Year>=1946 & Year<=2009)

## Italy uses another data series through 1946 and is excluded from GITD
    postwar until 1951
RR <- subset(RR, !(Year<1951 & Country=="Italy"))
```

Figure 5.2: Code snippet of the subset generation performed in the RR.R script.

```
  retrieveSubset <- function(pid, handle_url) {
    url <- sprintf("%s/api/handles/%s", handle_url, pid)
    resp <- GET(url)
    jsonRespo <- content(resp, as="parsed")

    for (x in jsonRespo$values){
      if(x$type == "API_URL") {
        resource_url <- x$data$value
      }
    }

    return(fromJSON(content(GET(resource_url), as="text"))
        $result$records_preview)
  }

  pid_file <- "pid"
  pid <- readChar(pid_file, file.info(pid_file)$size)

  RR <- retrieveSubset(pid, "http://localhost:8000")
```

Figure 5.3: Added code to RR_REST.R

2. A experiment that expects the source data to be located in `./countries_dataset.csv` (The experiment expects that before the dataset was retrieved using the ckanfetch CLI tool described in 4.3.1).

For the first version, the function `retrieveSubset` was added to the experiment code. It resolves a Handle PID and returns the subset. All lines of code that were added to the experiment (including the implementation of this function) as demonstrated in listing 5.3

For the second version, a wrapper script was created that handles the retrieval of the subset and starts the R script. This bash script is shown in listing 5.4

5.2.3 Generic Functional Test Case

In the evaluation project, each functional test case is declared as a separate Python class. As we were able to identify common steps that had to be done for each functional test case this similarities were factored out into a generic functional test case class.

```
#!/bin/bash

pid=$(head -n 1 pid)

ckanfetch -h http://localhost:8000 \
  --format csv \
  --csv-delimiter , \
  --include-header "$pid"

Rscript ./RR.R
```

Figure 5.4: Wrapper script for executing RR_CLI.R

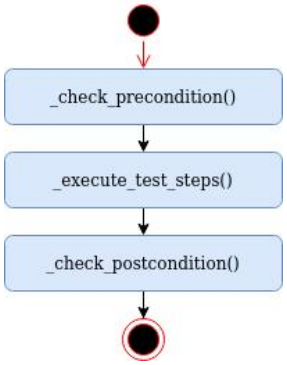


Figure 5.5: Flow diagram of a generic functional test case. The blue actions are abstract methods that have to be implemented by the specific test case implementations.

Each functional test consists of three steps:

1. check precondition,
2. execute test steps,
3. check postcondition.

For each specific functional test case, these steps have to be implemented. A visual representation of the execution flow is depicted in Figure 5.5. The blue action entities represent the methods that have to be implemented by the specific test cases.

5.2.4 Test Case Definitions

FTC 1.1 - Resource Publishing

Description In this test case, the dataset `countries_dataset.csv` is uploaded as a new resource to the CKAN repository. This will also cover the publication of the related metadata.

__check__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.

__execute__test__steps()

1. Create new Package 'Reinhard & Rogoff - Experiment'.
2. Upload `countries_dataset.csv` as new resource.
3. Wait 5 seconds for the datapusher to parse the csv file and push it to the datastore.

__check__postcondition()

- The CKAN package and resource are added to the solr index.
- The provided meta data is stored along with the uploaded package.
- The uploaded resource is stored in an MongoDB collection.
- The datastore_active flag is set to true for the created resource.

FTC 2.1 - Insert Record

Description This test case inserts a new record to an existing datastore resource.

__check__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The `countrie_dataset.csv` was uploaded to the CKAN repository.
- A record with id 1276 does not exist in the datastore resource.

__execute__test__steps()

1. Insert a new record (as defined in 5.6) into the datastore resource.

```
NEW_RECORD = {  'id': 1276, 'Country': 'Australia', 'Year': 2010,
        'Debt': 101136.25205, 'RGDP': None, 'GDP': None,
        'dRGDP': 0.732249739168633, 'GDPI': 109.15168,
        'GDP1': None, 'GDP2': 1201390, 'RGDP1': None,
        'RGDP2': 1100661, 'GDPI1': None, 'GDPI2': None,
        'Infl': '1.629', 'Debt1': None, 'Debt2': None,
        'Debtalt': None, 'GDP2alt': None, 'GDPalt': None,
        'RGDP2alt': None, 'debtgdp': 8.41826984160015,
        'GDP3': None, 'GNI': None, 'lRGDP': None,
        'lRGDP1': None, 'lRGDP2': 1092660 }
```

Figure 5.6: Record used for the insert operation in FC 2.1

__check__postcondition()

- The new record is added to the CKAN datastore.
- A timestamp was added to the record that represents the point of time when the record was added to the resource.
- The new record is visible when the current state of the datastore is queried.

FTC 2.2 - Modify Record

Description This test case modifies a record of an existing datastore resource.

__check__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The `countrie_dataset.csv` was uploaded to the CKAN repository.
- A record with id 1 already exists in the datastore resource.

__execute__test__steps()

1. Replace record with id 1 with a new record as defined in listing 5.7.

```
MODIFIED_RECORD = { 'id': 1, 'Country': 'Australia', 'Year': 2000,
          'Debt': None, 'RGDP': None, 'GDP': None,
          'dRGDP': None, 'GDPI': None, 'GDP1': None,
          'GDP2': None, 'RGDP1': None, 'RGDP2': None,
          'GDPI1': None, 'GDPI2': None, 'Infl': None,
          'Debt1': None, 'Debt2': None, 'Debtalt': None,
          'GDP2alt': None, 'GDPalt': None, 'RGDP2alt': None,
          'debtgdp': None,  'GDP3': None, 'GNI': None,
          'lRGDP': None, 'lRGDP1': None, 'lRGDP2': None }
```

Figure 5.7: Wrapper script for executing RR_cli.R

__check__postcondition()

- When retrieving the full dataset only the latest version of the record is included.
- The preceding version of the record is still stored in the MongoDB collection.
- A timestamp was added the the preceding version of the modified record that represents the point of time when the record was invalidated.
- The current version of the record was added as new MongoDB document to the collection including a timestamp that represents the point of time from which the current version of the record is valid.

FTC 2.3 - Delete Record

Description Delete records of an existing datastore resource.

__check__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The `countrie_dataset.csv` was uploaded to the CKAN repository.
- A record with id 1 exists within the datastore resource.

__execute__test__steps()

1. Delete record with id 1 from the datastore resource.

__check__postcondition()

- When retrieving the full dataset the deleted record is not included into the result set.
- A timestamp was added to the deleted record that represents the point of time when the record was deleted.

FTC 3.1 - Query By Value

Description In this test case, a query is submitted to the datastore that retrieves all records where a specific field exactly matches the provided parameter.

__check__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The `countrie_dataset.csv` was uploaded to the CKAN repository.

__execute__test__steps()

1. Perform a `datastore_search` operation with the `filter` statement `{"Country": "Austria"}`.

__check__postcondition()

- The results equal the expected results as defined in `data/expected_results/ftc3_1.json`.

FTC 3.2 - Range Query

Description In this test case, a query is submitted to the datastore that retrieves all records where a specific record field applies to a range of values.

__check__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The `countrie_dataset.csv` was uploaded to the CKAN repository.

__execute__test__steps()

1. Perform a `datastore_search` operation with the `filter` statement `{"Debt": ">200000000"}`.

__check__postcondition()

- The results equal the expected results as defined in `data/expected_results/ftc3_2.json`.

FTC 3.3 - Full Text Query

Description In this test case, a fulltext query is submitted to the datastore. This query retrieves all records where the query parameter occurs in one of the attributes with the datatype 'text'.

__check__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The `countrie_dataset.csv` was uploaded to the CKAN repository.

__execute__test__steps()

1. Perform a `datastore_search` operation with the `q` parameter `Aus`.

__check__postcondition()

- The results equal the expected results as defined in `data/expected_results/ftc3_3.json`.

FTC 3.4 - Query with Sort Order

Description In this test case, the dataset resource is sorted by a defined field in ascending order.

__check__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The `countrie_dataset.csv` was uploaded to the CKAN repository.

__execute__test__steps()

1. Perform a `datastore_search` operation with the `sort` parameter `Dept asc`.

__check__postcondition()

- The results equal the expected results as defined in `data/expected_results/ftc3_4.json`.

FTC 4.1 - PID Generation and Retrieval (Filter Statement)

Description In this test case, a persistent identifier is issued for a subset where a specific field exactly matches a value. Subsequently, it is tested if the retrieval of this subset stays stable after modification of the dataset in the CKAN repository.

__check__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The `countrie_dataset.csv` was uploaded to the CKAN repository.

__execute__test__steps()

1. An internal PID is issued for the filter statement `{"Country":"Austria"}`.
2. Wait 5 seconds.
3. Insert new record to the datastore resource.
4. Retrieve subset using the generated PID.
5. Calculate MD5 hash of the retrieved result.
6. Update an existing record.

7. Retrieve subset using the generated PID.
8. Calculate MD5 hash of the retrieved result.
9. Delete existing records.
10. Retrieve subset using the generated PID.
11. Calculate MD5 hash of the retrieved result.

__check__postcondition()

- All generated MD5 hash values should be equal.

FTC 4.2 - PID Generation and Retrieval (Range Query)

Description In this test case, a persistent identifier is issued for a subset where a specific field matches a range query. Subsequently, it is tested if the retrieval of this subset stays stable after modification of the dataset in the CKAN repository.

__check__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The `countrie_dataset.csv` was uploaded to the CKAN repository.

__execute__test__steps()

1. An internal id is issued for the filter statement `{"Debt":">200000000"}`.
2. Wait 5 seconds.
3. Insert new record to the datastore resource.
4. Retrieve subset using the generated PID.
5. Calculate MD5 hash of the retrieved result.
6. Update an existing record.
7. Retrieve subset using the generated PID.
8. Calculate MD5 hash of the retrieved result.
9. Delete existing records.
10. Retrieve subset using the generated PID.
11. Calculate MD5 hash of the retrieved result.

__check__postcondition()

- All generated MD5 hash values should be equal

FTC 4.3 - PID Generation and Retrieval (Full Text Query)

Description In this test case, a persistent identifier is issued for a subset where records match a fulltext query. Subsequently, it is tested if the retrieval of this subset stays stable after modification of the dataset in the CKAN repository.

__check__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The `countrie_dataset.csv` was uploaded to the CKAN repository.

__execute__test__steps()

1. An internal id is issued for the fulltext query `Aus`.
2. Wait 5 seconds.
3. Insert new record to the datastore resource.
4. Retrieve subset using the generated PID.
5. Calculate MD5 hash of the retrieved result.
6. Update an existing record.
7. Retrieve subset using the generated PID.
8. Calculate MD5 hash of the retrieved result.
9. Delete existing records.
10. Retrieve subset using the generated PID.
11. Calculate MD5 hash of the retrieved result.

__check__postcondition()

- All generated MD5 hash values should be equal.

FTC 4.4 - PID Generation and Retrieval (Query with Sort Order)

In this test case, a persistent identifier is issued for a subset which is ordered ascending by a defined field. Subsequently, it is tested if the retrieval of this subset stays stable after modification of the dataset in the CKAN repository.

__check__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The `countrie_dataset.csv` was uploaded to the CKAN repository.

__execute__test__steps()

1. An internal id is issued for the filter statement `{}` and a sort order `Dept asc`.
2. Wait 5 seconds.
3. Insert new record to the datastore resource.
4. Retrieve subset using the generated PID.
5. Calculate MD5 hash of the retrieved result.
6. Update an existing record.
7. Retrieve subset using the generated PID.
8. Calculate MD5 hash of the retrieved result.
9. Delete existing records.
10. Retrieve subset using the generated PID.
11. Calculate MD5 hash of the retrieved result.

__check__postcondition()

- All generated MD5 hash values should be equal.

FTC 5.1 - Retrieval of Persistently Identified Subset within Computation Experiment (REST API)

Description In this test case, an existing experiment code ist modified in a way that instead of loading the dataset form the local filesystem it is retrieved by the repository. To identify the needed subset, its PID is provided in the source code. Subsequently, it is verified that a modification of the dataset in the repository does not affect the results of the experiment

__check__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The `countrie_dataset.csv` was uploaded to the CKAN repository.

__execute__test__steps()

1. Issue a PID for query
   `{'Year': {'$gte': 1946, '$lte': 2009}, '$or': [{'Year': {'$not': {'$lt': 1951}}}, {'Country': {'$not': {'$eq': 'Italy'}}}]}`.
2. Run the experiment.
3. Calculate a MD5 hash value of the results.
4. Push source code to GitLab repository.
5. Tag version as `1.0`.
6. Assign handle PID to version `1.0`.
7. Create a new dataset for the experiment which contains both PIDs (code and data) and the associated metadata.
8. Modify the dataset in the CKAN repository.
9. Re-run the experiment.
10. Calculate a MD5 hash value of the results.
11. Purge repository.
12. Modify the source code in the GitLab repository.

13. Checkout the the source code using its PID.
14. Re-run the experiment.
15. Calculate a MD5 hash value of the results.

__check__postcondition()

- The MD5 hash values of the initial run and the re-execution are identical.
- The experiment dataset was created.
- The experiment dataset contains two resources referencing code and data.
- The experiment dataset provides metadata about the experiment.

FTC 5.2 - Retrieval of Persistently identified Subset within Computation Experiment (CLI Tool)

Description In this test case, the datasets used by an experiment are loaded to the local filesystem using the proposed CLI tool for retrieving persistently identified data subsets to the local file system before the experiment is executed. To identify the needed subset, its PID is provided. Subsequently, it is verified that a modification of the dataset in the repository does not affect the results of the experiment. Loading the dataset and execution of the experiment is handled by a wrapper bash script.

__check__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The `countrie_dataset.csv` was uploaded to the CKAN repository.

__execute__test__steps()

1. Issue a PID for query
   `{'Year': {'$gte': 1946, '$lte': 2009}, '$or': [{'Year': {'$not': {'$lt': 1951}}}, {'Country': {'$not': {'$eq': 'Italy'}}}]}`.
2. Run the experiment.
3. Calculate a MD5 hash value of the results.
4. Push source code to GitLab repository.

5. Tag version as `1.0`.
6. Assign handle PID to version `1.0`.
7. Create a new dataset for the experiment which contains both PIDs (code and data) and the associated metadata.
8. Modify the dataset in the CKAN repository.
9. Re-run the experiment.
10. Calculate a MD5 hash value of the results.
11. Purge repository.
12. Modify the source code in the GitLab repository.
13. Checkout the the source code using its PID.
14. Re-run the experiment.
15. Calculate a MD5 hash value of the results.

__check__postcondition()

- The MD5 hash values of the initial run and the re-execution are identical.
- The experiment dataset was created.
- The experiment dataset contains two resources referencing code and data.
- The experiment dataset provides metadata about the experiment.

5.2.5 Results

All functional test cases described in this section were executed on the default docker-compose setup. Table 5.5 shows that each test case passed the test run. By transforming the defined use cases to functional test cases, we have shown that the provided solution can cover all requirements defined by the use case diagrams.

5.3 Evaluation of Non-Functional Requirements

In this section, it is evaluated if the provided solution meets the non-functional requirements defined in Section 1.3. In general there are two major requirements that have to be evaluated:

Identifier	System Configuration	Result
1.1	default	✓
1.2	default	✓
2.1	default	✓
2.2	default	✓
2.3	default	✓
3.1	default	✓
3.2	default	✓
3.3	default	✓
3.4	default	✓
4.1	default	✓
4.2	default	✓
4.3	default	✓
4.4	default	✓
5.1	default	✓
5.2	default	✓

Table 5.5: Functional Test Report (✓ passed, ✗ failed)

- The provided solution should not have a worse run time complexity than the reference datastore implementation on queries to the current state of the datastore resource.
- The response time of the retrieval of persistently identified subsets of an dataset should not be affected by the number of subsequent modifications to the dataset.

Besides these non-functional requirements, this section tries to answer following questions:

- How does the query response time of the proposed solution compare to the default datastore implementation provided by CKAN?
- To what extent does the implemented versioning mechanism affect the query response time?
- Which measures can be taken to improve the query response time of the proposed solution?
- By what percentage do the suggested measure improve the query response time of the proposed solution?

These test cases do not define a postcondition. The collected results are discussed explicitly in Section 5.3.5

5.3.1 Test Assets

All non-functional tests are based on the `UC Berkeley Home IP Web Traces` dataset published by Steve D. Gribble[5]. As there is no persistent identifier assigned to the dataset, we published the original dataset on Zenodo[6]. Unfortunately, this dataset is in a binary format which requires a proprietary tool[7] for decoding (which we also published on Zenodo[8] in the course of this evaluation). Therefore, we did not only publish the original dataset but also converted the binary file into a plain text csv file which was published as a own deposit. The following Section 5.3.2 describes in detail the required modifications to the original files.

5.3.2 Test Asset Preprocessing

The `UC Berkeley Home IP Web Traces` consists of five gzip archives:

- UCB-home-IP-846890339-847313219.tr.gz
  (from `Fri Nov 1 15:18:59 1996` through `Wed Nov 6 12:46:59 1996`
- UCB-home-IP-847313219-847601221.tr.gz
  (from `Wed Nov 6 12:46:59 1996` through `Sat Nov 9 20:47:01 1996`
- UCB-home-IP-847601221-848004424.tr.gz
  (from `Sat Nov 9 20:47:01 1996` through `Thu Nov 14 12:47:04 1996`
- UCB-home-IP-848004424-848409417.tr.gz
  (from `Thu Nov 14 12:47:04 1996` through `Tue Nov 19 05:16:57 1996`
- UCB-home-IP-848278026-848292426.tr.gz
  (4 hour snippet, not used for this non-functional requirements evaluation)

For transforming a binary dataset into plain text, the `showtrace` command is provided within the provided tools. Before using, the tool has to be compiled by calling the included make file (as we are running the experiment in a linux system we had to use the according `LIBS` variable as shown in Figure 5.8).

Having compiled the `showtrace` command, we were able to transform the source dataset into plain text by calling `zcat $UCB_ARCHIVE | showtrace` as described in the original documentation of the dataset. To end up having one file containing all records, we piped the output of each command execution into the same file. Unfortunately this was not sufficient, as the `showtrace` command included following not printable characters into the resulting file:

[5] `ftp://ita.ee.lbl.gov/html/contrib/UCB.home-IP-HTTP.html` (visited on: 10/13/2020)
[6] `https://doi.org/10.5281/zenodo.4020425` (visited on: 10/13/2020)
[7] `ftp://ita.ee.lbl.gov/software/UCB-home-IP.tools.tar.gz` (visited on: 10/13/2020)
[8] `https://doi.org/10.5281/zenodo.4020422` (visited on: 10/13/2020)

```
23   # Uncomment the following for Solaris
24   #LIBS = -lsocket -lnsl -lintl -ldl -lm
25
26   # Uncomment the following for Linux
27   LIBS = -ldl -lm
```

Figure 5.8: Snippet of the make file used to compile the showtrace command.

```
1  import sys
2
3  skipped = 0
4
5  with open(sys.argv[1], 'rb') as input:
6    with open('preprocessed_{}'.format(sys.argv[1]), 'wb') as output:
7      buf = input.read(1)
8
9      while buf:
10       if buf[0] == 0x1e:
11         output.write(b' ')
12       if buf[0] == 0x19 or buf[0] == 0x17:
13         skipped += 1
14       else:
15         output.write(buf)
16       buf = input.read(1)
17
18 print('{0} characters skipped'.format(skipped))
```

Figure 5.9: Preprocessing script to handle not printable characters from a plaintext file.

- `0x17`
- `0x19`
- `0x1e`

As these characters caused problems when reading the result file, they were removed by a script shown in Figure 5.9. This script is located in the root folder of the `ckanext-mongodatastore-evaluation` project. Finally, all spaces were replaced by a semicolon. The final result of the pre-processing steps was on published on Zenodo as well[9].

5.3.3 Generic Non-Functional Test Case

In the evaluation Python project each non-functional test case is declared as a separate class. As we were able to identify common steps that had to be done for each non-

[9] `https://doi.org/10.5281/zenodo.4058379` (visited on: 10/13/2020)

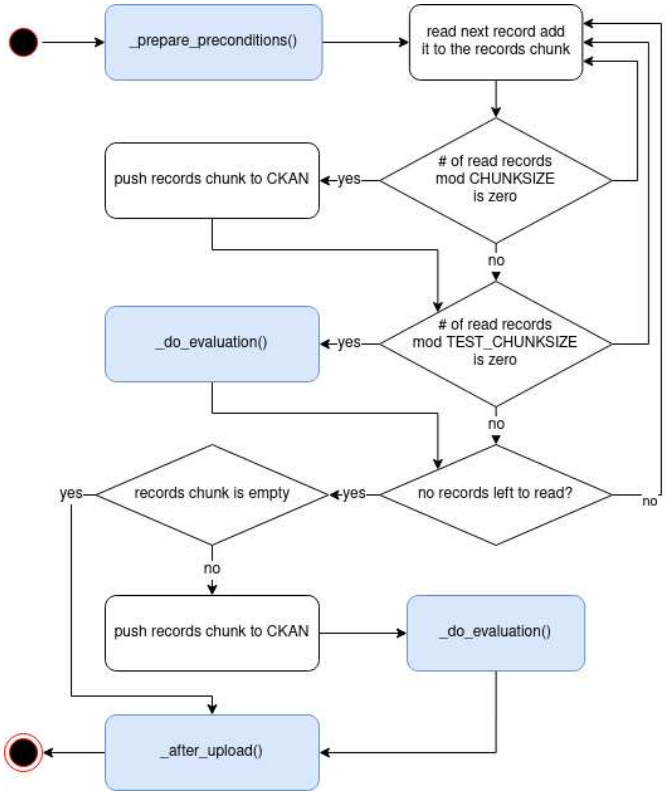


Figure 5.10: Flow diagram of a generic non-functional test case. The blue actions are abstract methods that have to be implemented by the specific test case implementations.

functional test case, these steps were factored out into a generic non-functional test case class.

In each non-functional test case, a large test dataset is uploaded in chunks to a CKAN resource. In-between the upload of these chunks, the individual evaluation tasks are executed. Therefore, it is possible to measure how certain metrics (e.g. query response time) are affected by the size of a dataset. It is also possible to define actions that have to be done once the whole dataset is uploaded.

For each test case, a chunk size and test chunk size have to be defined. The chunk size defines how many records are uploaded at each upload action and the test chunk size defines how many records are uploaded between the execution of the evaluation steps. The full flow chart of the execution of non-functional test cases is depicted in Figure 5.10. The blue action entities represent the methods that have to be implemented by the specific test cases.

5.3.4 Test Case Definitions

NFTC 1 - Examine the Query Response Time on Current State

Description This test uploads the "UC Berkeley Home IP Web Trace" dataset in chunks of 500000 records. After each chunk, an evaluation phase is triggered where randomly generated queries are submitted to the datastore. The goal of this test case is to examine the response time behaviour of queries to the current state of the dataset in terms of dataset size.

__prepare__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The package `ucbtrace` was created.

__do__evaluation()

1. Submit the randomly generated filter queries to the CKAN datastore and measure the duration of the individual calls.
2. Submit the defined fulltext queries to the CKAN datastore and measure the duration of the individual calls.
3. Write average query durations for filter and fulltext queries to the result file.

NFTC 2 - Examing the Query Response Time on Non-Versioned MongoDB Collections

Description This test uploads the "UC Berkeley Home IP Web Trace" dataset in chunks of 500000 records. After each chunk, an evaluation phase is triggered where randomly generated queries are submitted to the datastore. The difference to NFTC 1 is, that we call the `nv_query` (i.e. a non-versioned query operation) instead of `datastore_search`. This non-versioned query operation behaves the same way as the datastore_search operation with the difference that it does not include the condition for only querying the latest state. This allows us to measure how fast MongoDB can handle queries without the versioning overhead as the whole collection represents the current state of the dataset. The goal of this test case is to examine the impact of the implemented record versioning mechanism on the query run time.

__prepare__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The package `ucbtrace` was created.

__do__evaluation()

1. Submit the randomly generated filter queries to the CKAN datastore and measure the duration of the individual calls (using the non-versioned query action).
2. Submit the defined fulltext queries to the CKAN datastore and measure the duration of the individual calls (using the non-versioned query action).
3. Write average query durations for filter and fulltext queries to the result file.

NFTC 3 - Examine Retrieval Time of Stored Queries

Description This test uploads the first 1000000 records of the "UC Berkeley Home IP Web Trace" dataset to a CKAN datastore. Based on the first million of records, 10 PIDs are issued (random queries). Then the remaining records are pushed to the datastore in chunks of 500000 records. After each chunk, the time for retrieving the persistently identified data subsets is measured

__prepare__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The package `ucbtrace` was created.
- The `preprocessed_trace_1000000` file was uploaded to the CKAN resource.

__do__evaluation()

1. Retrieve randomly generated stored filter queries and measure the duration of the individual calls.
2. Retrieve randomly generated stored fulltext queries and measure the duration of the individual calls.
3. Write average query durations for filter and fulltext queries to the result file.

Identifier	Configurations
NFTC 1	default, legacy
NFTC 2	default
NFTC 3	default
NFTC 4.1	default, 4shards, 5shards, 6shards
NFTC 4.2	legacy

Table 5.6: Summary of which non-functional test case was executed on which docker-compose configuration

NFTC 4.1/4.2 - Examine the Impact of Indexing

Description This test case examines the performance impact of indices on the query response time. NFTC 4.1 and 4.2 examin the performance improvements in MongoDB and PostgreSQL respectively.

__prepare__precondition()

- An evaluation user with API key `302b24d4-8a23-47bd-baef-b8e8236d27a3` exists.
- The organization `tu-wien` exists.
- The package `ucbtrace` was created.

__after__upload()

1. Submit the randomly generated filter queries based on the field `client_port` to the CKAN datastore and measure the duration of the individual calls.
2. Delete index on `client_port`.
3. Submit the randomly generated filter queries based on the field `client_port` to the CKAN datastore and measure the duration of the individual calls.
4. Write average query durations for queries with and without index to the result file.

5.3.5 Results

Some of the non-functional test cases were executed several times on different docker-compose configurations, e.g. NFTC 1 was executed on the default and legacy configuration in order to be able to compare the performance of the new solution to the default datastore implementation provided by CKAN. An overview of which test case was executed on which docker-compose configuration is given in Table 5.6.

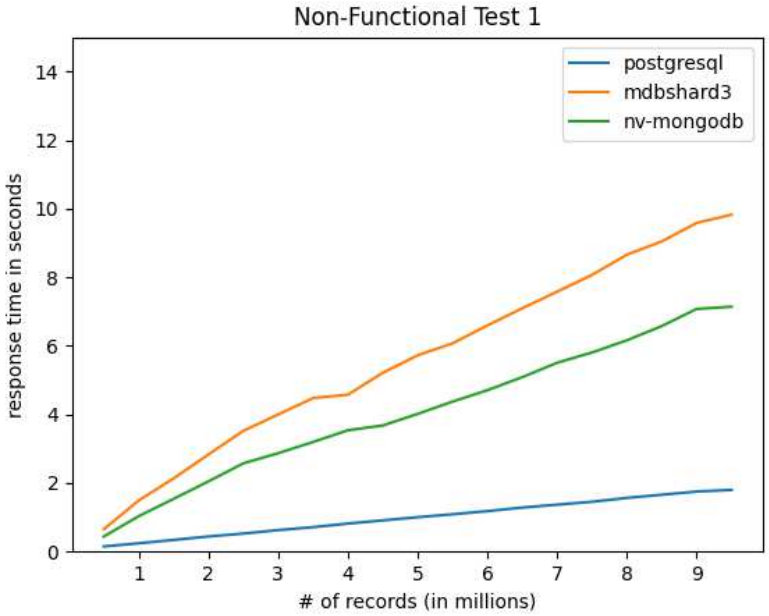


Figure 5.11: Results of the NFTC 1 experiment on filter queries

Figure 5.11 shows the query run time for the proposed solution, a non-versioned MongoDB collection and the default datastore implementation based on PostgeSQL. The proposed solution as well as the default datastore have a linear query run time complexity, with the difference that the default datastore performed five times faster.

By comparing the proposed solution's performance to the non-versioned result, we can see that without the versioning overhead MongoDB already performs worse than the solution based on PostgreSQL. Adding the versioning overhead leads to an increase of the query run time by 37% in the case of filter queries.

In case of fulltext queries (depicted in Figure 5.12), the query run time of the provided solution exceeds the results of the PostgreSQL based solution by 25% and similar to the filter queries the non-versioned MongoDB collection performs worse than the default datastore implementation provided by CKAN. Compared to the non-versioned MongoDB collection, the run time of the proposed solution is 25% longer.

Figure 5.13 and 5.14 (for stored filter and fulltext queries respectively) show how the retrieval run time of stored queries behave when additional records are added to the datastore. In both cases, the retrieval run time remained stable when additional records were added to the datastore.

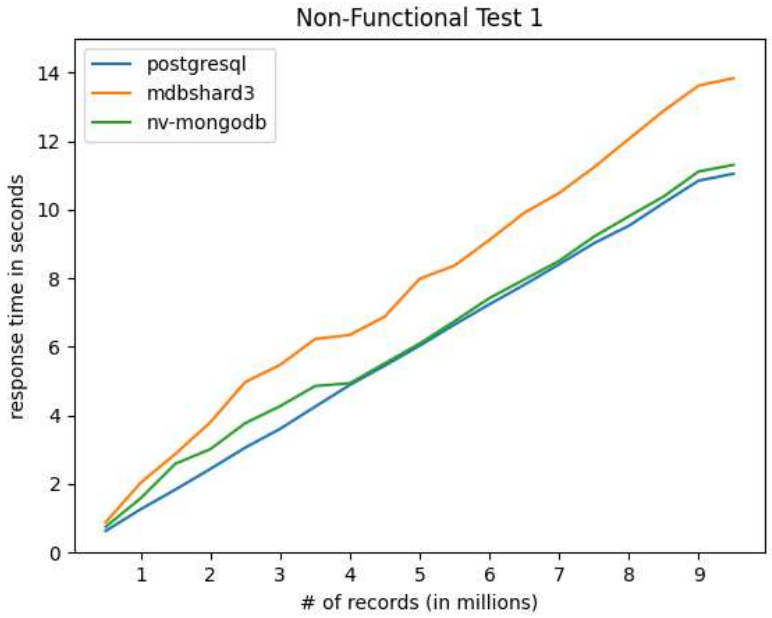


Figure 5.12: Results of the NFTC 1 experiment on fulltext queries

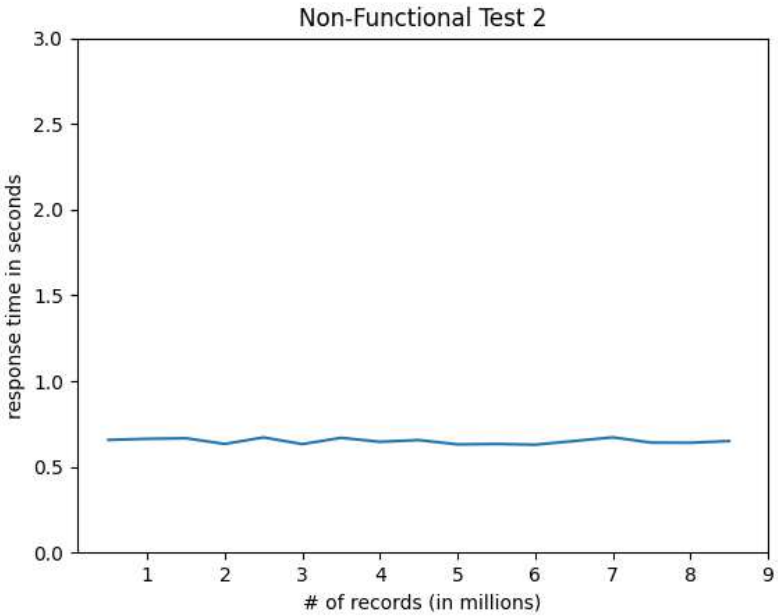


Figure 5.13: Results of the NFTC 2 experiment on filter queries

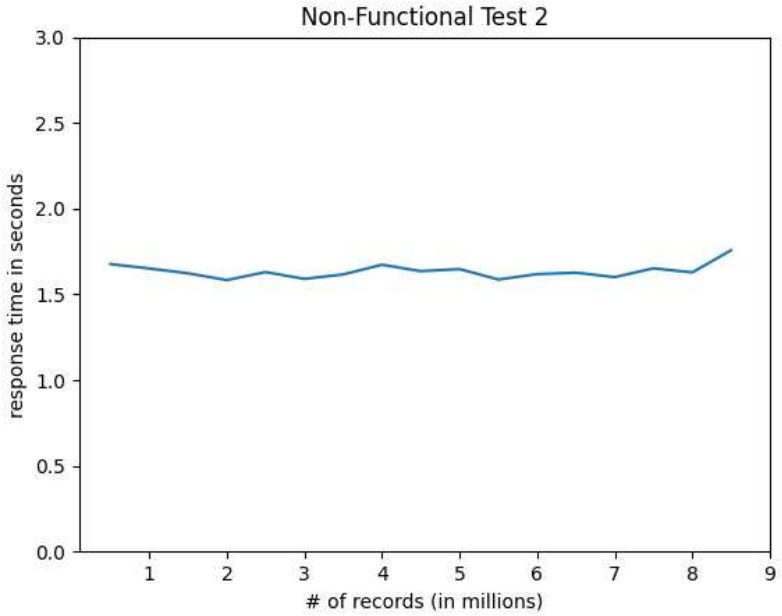


Figure 5.14: Results of the NFTC 2 experiment on fulltext queries

5.3.6 Performance Improvements

Evaluation Scenario NFTC 1 demonstrated that the already existing datastore implementation scales better on large datasets than the suggested solution.

This subsection will cover two approaches to improve the performance of MongoDB queries. The first approach is horizontal scaling, i.e. increasing the number of shards. The second approach seeks to optimize response times through targeted usage of indices.

Horizontal Scaling

For a better handling of higher loads (increasing dataset sizes, higher query rates, etc.) two methods exist: horizontal and vertical scaling. Vertical scaling focuses on improving the hardware capacity of the existing instance (e.g. increasing memory size or using a more powerful CPU) which is limited by the available technology. On the other hand, horizontal scaling pursues the strategy of splitting up the database in smaller instances in order to parallelize the workload.

In order to evaluate to what extent the usage of horizontal scaling the run time of queries, the fourth non-functional test case (NFTC 4.1) was executed on four different MongoDB setups (using three, four, five, and six shards). The results were compared to the results of NFTC 4.2 executed on the legacy setup.

Performance Optimization by Indices Usage

Similar to relational databases MongoDB also supports the usage of indices. An index stores partial data of a collection (or a relational database table respectively) in a way that makes filtering or sorting the data more efficient. In MongoDB, indices are based on B-Tree structures. The same structures are also used in PostreSQL databases by default [10,11].

As a goal of this thesis is to keep the solution as generic as possible, indices are only applied on fields, that are common for every collection (e.g. index on `_valid_to` field). Still the CKAN datastore API interface provides the possibility to define indices on specific fields of created resources and in some cases a researcher may already know in advance that certain fields may be used frequently as a filter criteria.

MongoDB supports different types of indices, e.g. Single Field Index, which are described in the database system's documentation [12]. As the name suggests, the Single Field Index stores the values of a particular field (e.g. `client_port`) in a B-Tree which enables the database to efficiently find documents that meet filter criterias containing the field.

In order to evaluate to what extent the usage of indices affects the run time of queries, the third non-functional test case (**NFTC 3**) performs queries (on the field `client_port`) on a collection with and without index.

Result

To examine the impact on query run times, a randomly generated set of queries on the field `client_port` is performed on the full *UC Berkeley Home IP Web Trace* with and without an index defined on the field `client_port`. This test case was repeated on a setup with four, five, and six shards and finally it was also conducted on the legacy setup in order to compare the results to the datastore implementation based on PostgreSQL.

Figure 5.15 shows the result of each test run. The usage of indices decreases the query run time by 83 % on each shard setup. By doubling the amount of shards, a run time reduction by 25% can be achieved. Still, the solution based on PostgreSQL can achieve a better run time with and without index.

5.4 Discussion on the Results of Functional and Non-Functional Tests

As expected, the execution of all functional test cases passed. In conjunction with the fact that for each use case at least one corresponding functional test case was implemented (as shown in Table 5.4) we can conclude that the proposed solution provides the required functionalities.

[10] `https://docs.mongodb.com/manual/indexes/` (visited on: 10/13/2020)
[11] `https://www.postgresql.org/docs/11/view-pg-indexes.html` (visited on: 10/13/2020)
[12] `https://docs.mongodb.com/manual/indexes/` (visited on: 10/13/2020)

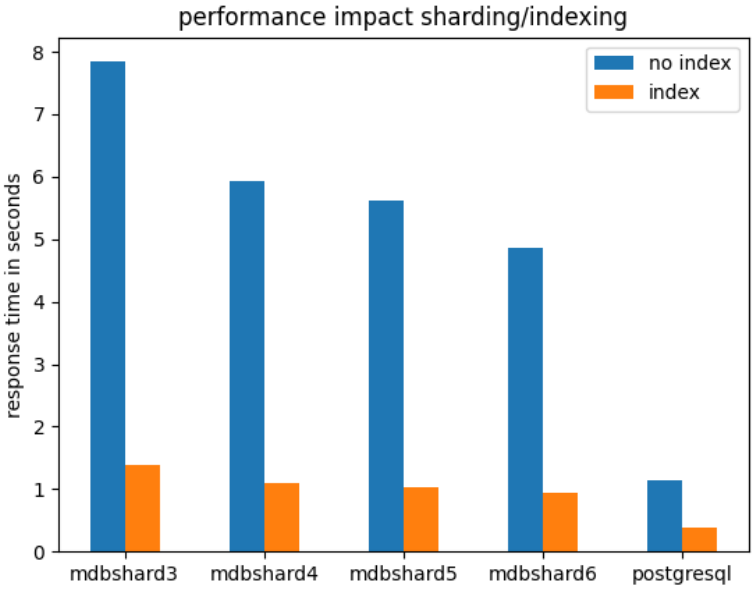


Figure 5.15: Accommodated results of the NFTC 3 experiment

Comparing the results of the test case NFTC1 we observed that compared to the datastore implementation based on PostgreSQL the proposed solution has an average response time which is 5 times longer. This observation was expected because unlike the default datastore implementation the proposed solution has to deal with a versioning overhead.

To find out by what fraction the versioning affects the query response time we also performed the evaluation on a non-versioned MongoDB collection. The results show that, speaking of a collection containing 9,244,728 records, the response time of the versioned datastore exceed the response time of the PostgreSQL datastore by 8,0 seconds and the non-versioned datastore by 5,3 seconds. This demonstrates that the versioning is only accountable for 34% of the increase in run time.

The first approach to decrease the response time was horizontal scaling (i.e. sharding the database into 4,5 and 6 MongoDB shards). As expected, the response time of queries on a 6-shard setup decreased by 25% compared to the 3-shard setup. The second approach, using an index, resulted in a response time reduction by 82 to 80%.

Nevertheless, for using indices the creator of the datastore has to know in advance which fields are most likely used when querying the dataset. Sharding, on the other hand, is a more generic way to improve the performance of the proposed solution, as it does not require any information on how the data structure looks like or which fields are most likely used in queries.

Recommendation	Compliance
R1 - Data Versioning	✓
R2 - Timestamping	✓
R3 - Query Store Facilities	✓
R4 - Query Uniqueness	✓
R5 - Stable Sorting	✓
R6 - Result Set Verification	✓
R7 - Query Timestamping	✓
R8 - Assigning Query PID	✓
R9 - Storing the Query	✓
R10 - Automated Citaiton Text Generation	✓
R11 - Landing Pages	✓
R12 - Machine Actionability	✓
R13 - Technology Migration	✗
R14 - Migration Verification	∼

Table 5.7: Summary of the proposed solution's compliance to the *RDA Recommendations for Data Citation* (✓ fully complies, ∼ partially complies, ✗ does not comply

5.5 Compliance with RDA Recommendations for Data Citation

This section examines to what extent the *RDA Recommendations for Data Citation* are covered by the proposed solution. Table 5.7 summarizes to what extent each recommendation was implemented in the proposed solution. The further subsections provide a discussion for each recommendation on how the recommendation were implemented or why they are out of scope of this thesis. The beginning of each subsection lists the exact wording of the recommendation as published in "Identification of Reproducible Subsets for Data Citation, Sharing and Re-Use" [18].

5.5.1 Data Versioning (R1)

"*Recommendation*: Apply versioning to ensure earlier states of data sets can be retrieved."

As described in Section 4.1.2 each record is enhanced by several meta fields. The fields `_created` and `_valid_to` store the information when a record was added and to which point of time it was valid. The second field is optional, in case it is not present the record is still valid. Therefore, it is possible to retrieve the state of a dataset at an arbitrary point of time by filtering all records that were valid at this specific point of time. Therefore, this recommendation is fully covered by the provided implementation.

5.5.2 Timestamping (R2)

"*Recommendation*: Ensure that operations on data are timestamped, i.e. any additions, deletions are marked with a timestamp."

As described in Section 4.1.2 each insert and update operation results into a new inserted document (i.e. the new version of the record). For each record two timestamps are stored. The `_created` field stores the point of time when the document was created and the `_valid_to` field stores the point of time until this document was the latest version of a specific record. Delete operations only set the `_valid_to` field to the current point of time instead of deleting the document.

Although there is no timestamped history which operations were executed upon the datastore at which point of time, each document stores the information when it was created by an insert operation and when it was deleted or updated (i.e. until which point of time this specific version of a record was valid). As demonstrated in the following evaluation scenarios, this information is sufficient to retrieve earlier states of a dataset at an arbitrary point of time.

5.5.3 Query Store Facilities (R3)

"*Recommendation*: Provide means for storing queries and the associated metadata in order to re-execute them in the future."

As described in Section 4.1.4 the implementation maintains a separate database where all information related to queries that are submitted to the datastore a stored. This includes the query itself and all metadata that is needed to re-execute the query at a later point in time and information describing the query and its result set (e.g. fixity information, metadata about the author, etc.). As demonstrated in the following evaluation experiments, the stored queries provide enough information to persistently re-execute the queries that are submitted to the datastore.

5.5.4 Query Uniqueness (R4)

"*Recommendation*: Re-write the query to a normalised form so that identical queries can be detected. Compute a checksum of the normalised query to efficiently detect identical queries."

This recommendation is addressed by Section 4.1.1. Each query is represented as a JSON object. Ordering the attributes of the JSON queries by alphabet has no impact on their semantics. This ordered representation of the queries is used as the normalized query representation.

Each query entry in the quer ystore also contains a MD5 hash value of the normalized query. As queries are transformed into a normalized form and the hash value of the normalized queries is stored to the query store, this recommendation is covered by the provided implementation.

5.5.5 Stable Sorting (R5)

"*Recommendation*: Ensure that the sorting of the records in the data set is unambiguous and reproducible."

To guarantee stable sorting, each query is at least sorted by the unique internal id (i.e. _texttt_id).

5.5.6 Result Set Verification (R6)

"*Recommendation*: Compute fixity information (also referred to as checksum or hash key) of the query result set to enable verification of the correctness of a result upon re-execution."

The query store data model (as described in Section 4.1.4) also includes a hash value of the query's result set. Due to the fact that the calculation of this hash value may be time intensive for larger result sets this value is calculated asynchronously.

To make sure that every query that is persistently identified by a Handle PID also includes a result set hash value, the PID is assigned once the result set calculation is done.

5.5.7 Query Timestamping (R7)

"*Recommendation*: Assign a timestamp to the query based on the last update to the entire database (or the last update to the selection of data affected by the query or the query execution time)."

To meet this requirement the current timestamp is measured at the beginning of the query operation and all subsequent operations are related to this timestamp. This timestamp is also stored to the column `timestamp` to the `QUERY` table of the query store database.

5.5.8 Assigning Query PID (R8)

"*Recommendation*: Assign a new PID to the query if either the query is new or if the result set returned from an earlier identical query is different due to changes in the data. Otherwise, return the existing PID of the earlier query to the user."

As described in Section 4.1.6 PIDs are assigned asynchronously as the result set hash is required to distinguish if a new PID has to be issued for the query or an already existing PID can be reused in case an entry with an identical query hash and result set hash does already exist in the query store.

Due to the fact that the calculation of the result set hash can be a time intensive task for larger result sets, this task is executed as a asynchronous job. Once the calculation is done the datastore decides whether a new PID has to be issued or not.

5.5.9 Store the Query (R9)

"*Recommendation*: Store query and metadata (e.g. PID, original and normalised query, query and result set checksum, timestamp, superset PID, data set description, and other) in the query store."

As described in Section 4.1.4, the query, along with all necessary meta data, is stored in a dedicated query store database.

5.5.10 Automated Citation Text Generation (R10)

"*Recommendation*: Generate citation texts in the format prevalent in the designated community for lowering the barrier for citing and sharing the data. Include the PID into the citation text snippet."

Each stored query can be re-executed by visiting its landing page. This landing page also includes a generated citation text as described in Section 4.1.8. To provide a solution that can adapt easily to a communities citation format standards, the generated text is based on a HTML template which can be easily exchanged or modified.

5.5.11 Landing Pages (R11)

"*Recommendation*: Make the PIDs resolve to a human readable landing page that provides the data (via query re-execution) and metadata, including a link to the superset (PID of the data source) and citation text snippet."

For each stored query, the CKAN datastore implementation provides a landing page that contains all metadata of the query and a possibility to download the subset as csv file. As CKAN does not support the automatic assignment of PIDs to each uploaded resource out of the box for the current version of the datastore plugin no PID for the superset can be provided at the landing page. Except this shortcoming all related metadata values (timestamps, hash values, etc.) that are stored in the query store are displayed on the landing page.

The landing page can be accessed by resolving querie's Handle PID with at the global handle registry (i.e. `http://hdl.handle.net/`) which redirects the request directly to the landing page hosted by the CKAN datastore under the endpoint `/view_query`.

5.5.12 Machine Actionability (R12)

"*Recommendation*: Provide an API / machine actionable landing page to access metadata and data via query re-execution."

Each Handle PID entry gnerated by the CKAN datastore also includes a URL tagged as `API_URL`. A request to this URL retrieves a JSON object which includes the same information that is displayed at the landing page. The endpoint that provides this JSON object is hosted by the CKAN datastore under `/querystore_resolve`

5.5.13 Technology Migration (R13)

"*Recommendation*: When data is migrated to a new representation (e.g. new database system, a new schema or a completely different technology), migrate also the queries and associated fixity information."

As this is an organizational measure, this recommendation is not covered by this thesis which focuses on providing a repository solution. Nevertheless, the provided solution also provides a commandline tool which re-executes each query in the query store to verify if all hash values are still valid.

5.5.14 Migration Verification (R14)

"*Recommendation*: Verify successful data and query migration, ensuring that queries can be re-executed correctly."

This also describes an organizational measurement which was not directly addressed by this thesis but the datastore solution also comes with a commandline tool that re-executes each query in the query store and verifies if the result set hash values are still valid. A more detailed description of the tool is given in Section 4.1.9.

5.6 FAIR Guiding Principles Compliance

In the last two functional test cases (**FTC 5.1** & **FTC 5.2**) the experiment code from "Does high public debt consistently stifle economic growth? A critique of Reinhart and Rogoff" was published to CKAN along with the data it relies on using persistent identifiers. In this section we will evaluate to what extent this proposed way of publishing computational experiments along with the data subset it relies on aligns with the FAIR Guiding Principles. An overview of the result is given in Table 5.8.

5.6.1 Findability

The first findability principle (F1) - that is assigning a unique and persistent identifier to the datasets and related meta information - is fullfilled as both the source code and the dataset are assigned to an persistent identifier.

The second principle (F2) requires published data to be described with rich metadata. This requirement is covered as well. The CKAN dataset which includes the PIDs stores metadata information about the experiment (e.g. author, maintainer, license, etc.). In addition, the landing page of the persistently identified data subset contains metadata about the source resource, its fields, and the metadata required by the RDA Recommendations for Data Citation (timestamp of query submission, hash values of result set and query, etc.)

The third principle demands that published metadata contain a persistent identifier of the data it describes. This principle is fulfilled as the PIDs are also attached as key-value metadata fields to the CKAN dataset.

Principle	Institutional Website	CKAN
Findable		
F1 - Metadata is assigned to PID	✗	✓
F2 - Data is described with Metadata	✗	✓
F3 - Metadata includes an identifier of the data it describes	✗	✓
F4 - Metadata is indexed or registered in a search index	✗	✓
Accessible		
A1 - Usage of a standardized Communication Protocol	✓	✓
A1.1 - The Protocol is open, free and universally implementable	✓	✓
A1.2 - The protocol allows authentication and authorization	✓	✓
A2 - Metadata of deleted data is accessible	✗	∼
Interoperable		
I1 - Usage of a formal, accessible, shared and broadly applicable representation	✗	✓
I2 - Usage of Vocabulary that aligns with FAIR Principles	✗	✓
I3 - Metadata include qualified references to other (meta)data	✗	✓
Reusable		
R1 - Metadata are richly described	✗	∼
R1.1 - Meta(data) is released witha data usage license	✓	✓
R1.2 - Meta(data) is published with detailed provenance	✗	∼
R1.3 - Meta(data) meet domain-relevant community standards	✗	∼

Table 5.8: Summary of the proposed solution's compliance to the *FAIR Guiding Principles* (✓ fully complies, ∼ partially complies, ✗ does not comply)

CKAN uses a *Solr* search engine, therefore the registration of the datasets to a search index (F4) is already handled by the repository.

The old approach to publish the experiment does not fulfill any of these principles, as no explicit metadata about the published dataset and source code is stored along with the research assets.

5.6.2 Accessibility

According to the FAIR principles, making datasets accessible requires communication via standardized protocols. This requirement is specified in principle A1. In the context of this thesis, the only communication protocol between the user and the repository is HTTP or HTTPS (depending on the configuration of the webserver), therefore this requirement is satisfied as well.

Subitems 1 and 2 are fulfilled as HTTP and HTTPS are open standards and support authentication and authorization.

Due to the fact that all metadata is stored in the query store and not only in the repository, the information remains available even if the datasets are removed from the repository. This only applies to the metadata of the persistently identified dataset. For the metadata attached to the CKAN dataset which contains both PIDs this is not implemented yet, therefore this principle is partially fullfilled.

The old approach uses HTTPS as well and therefore A1, A1.1 and A1.2 are fulfilled. As the assets are not described with metadata at all, A2 is not fulfilled.

5.6.3 Interoperability

CKAN only allows to attach freetext key value pairs to datasets in order to describe datasets. It does not enforce any metadata model and therefore none of the *Interoperability* principles are fulfilled out-of-the-box.

By installing the CKAN extension `ckanext-dcat`, the metadata of each dataset is provided in a variety of formats (e.g. JSON-LD). Using this extension in combination of implementing an according profile (i.e. a mapping between the CKAN dataset and the according DCAT vocabulary), all *Interoperability* principles can be fulfilled.

The old approach does not fulfill any of the interoperability principles as the research assets are not described with metadata.

5.6.4 Reusability

CKAN allows users to attach licenses to datasets, therefore R1.1 is fulfilled. Nevertheless, CKAN does not provide a way to enforce metadata models by default (metadata is only described as textual key-value pairs), therefore the remaining principles (metadata are richly described, meta(data) is published with detailed provenance, meta(data) meet domain-relevant community standars) are only partially fulfilled.

The landing page of the old approach includes a license for the research assets, therefore R1.1 is fulfilled. Nevertheless, due to missing descriptive metadata R1, R1.2 and R1.3 are not fulfilled.

5.7 Summary

In this chapter, we examined to what extend the provided solution fulfilled all requirements defined in Chapter 1. First, we specified the system on which all evaluation experiments were conducted and how to re-execute them in Section 5.1.

By transforming the identified use cases into automated functional test cases the extent to which the provided solution can cover all use cases was examined in Section 5.2. In the following Section 5.3 performance issues and ways to improve the performance of the provided solution were discussed.

In the remaining Sections 5.5 and 5.6, we examined to which extend the provided solution aligns with the RDA Recommendations for Data Citation and the FAIR principles.

CHAPTER 6

Conclusion and Future Work

Research Data Management for computational experiments and their underlying datasets is a topic of emerging importance. The motivating examples listed in Section 1.1 show that publishing source code of computational experiments and the data they use on institutional websites without persistently citing the version of the code or the exact subset of used datasets makes reproducing those results cumbersome. Furthermore, this research data management practice makes it hard to find and reuse this assets. By introducing a new ecosystem consisting of a research data (CKAN) and code repository (GitHub) which aims to support researchers in publishing their computational experiments in a way that the experiments retrieve their datasets from a persistent datasource. Another goal of those experiments was that they can be easily found and re-used by the scientific community.

By implementing a new datastore plugin which aligns with the RDA Recommendations for Data Citation in CKAN, we demonstrated that it is possible to enable a research data repository to persistently identify subsets of the datasets it hosts.

In the next section, we revisit the research questions and conclude our findings by providing guidelines for improving FAIRness of computational experiments that rely on continuously evolving data.

6.1 Research Questions Revisted

Q1: To what extent can we automate persistent identification of data for continuously evolving databases used in scientific experiments?

- *How can continuously evolving data be stored in a versioned way in order to retrieve older states of the dataset?*

In order to being able to track all changes of a dataset, we introduced the metafields `_created` and `_valid_to` for each record that is stored in the datastore. Instead of deleting records or modifying records in the collection, the current version of a record is invalidated by setting the `_valid_to` to now and create a new record with the same logical id. As for each record it is defined when it was valid (from `_created` to `_valid_to`) it is possible to reconstruct the collection's state at any point in time.

– *What components need to be added to a research data repository to be capable of storing the required data for re-execution of historic queries in a reproducible way?*

  To add the capability of retrieving historic queries from the datastore, the RDA Recommendations were incorporated into the new CKAN datastore implementation. This did not only involve the implementation of a versioned datastore, but also setting up a query store.

– *To what extent does the versioning overhead affect the queries on the current state of the resource in terms of query run time?*

  A comparison between the query performance of versioned and non versioned collections revealed that versioning is accountable for 27% (filter queries) and 18% (fulltext queries) of the total query response time.

– *How is the query run time affected by the number of records stored in a repository resource?*

  We observed a linear relationship between the number of added records and increase in response time. Depending on the number of shards used, the gradient can be altered (an increase in number of shards decreases the gradient).

– *To what extent is the time for retrieving persistently identified subsets affected by the number of updates that are performed after the persistent identifier was issued?*

  We observed that the number of updates performed on a dataset after a PID is issued for a certain query does not affect the duration it takes to retrieve the persistently identified subset.

Q2: How can FAIRness of computational experiments based on continuously evolving data be increased?

- *How can persistently identified data subsets be published in a research data repository?*

  As demonstrated in FTC 5.1 and FTC 5.2 persistently identified data subsets can be published in the a repository by attaching a URL resource which contains the PID URL (e.g. `https://hdl.handle.net/PREFIX/xxxxxxx`) to a new repository deposit.

- *How can persistently identified versions of a source code repository be published in a research data repository?*

  As demonstrated in FTC 5.1 and FTC 5.2, persistently identified source code assets can be published in a repository by attaching a URL resource which contains the PID URL
  (e.g. `https://hdl.handle.net/PREFIX/xxxxxxx`) to a new repository deposit.

In Section 5.8 we evaluated that the proposed way of publishing computational experiments does improve FAIRness. Therefore, the general answer to **RQ2** is a set of guidelines we defined based on our proposed way of publishing computational experiments:

Guidlines for enhancing FAIRness of computational experiments that rely on subsets of continuously evolving datasets

1. Publish the source dataset in a repository that has persistent identification of subsets for continuously evolving data in place.

2. In case the dataset was already retrieved, check if the hash values are equal. Otherwise retrieve the dataset via PID at the beginning of the experiment code (e.g. using the proposed CLI tool *ckanfetch*).

3. Publish experiment source code in a source code versioning system that is capable of tagging specific versions of the code.

4. Assign PID to a specific tag of the source code (e.g. using the GitHub integration for Zenodo).

5. Use both PIDs when citing the experiment.

6.2 Future Work

This thesis provided a solution to persistently identify subsets continuously evolving datasets for usage in computational experiments. Throughout the evaluation we were able to identify two topics that are suitable for follow-up contributions.

First, the investigation of performance issues is a potential topic. The thesis evaluated the conceptual design based on MongoDB as this technology is the most popular NoSQL database in this segment. A follow-up work can be an evaluation on how different NoSQL technologies (e.g. Couchbase, Apache HBase, Reddis) perform in the context of this thesis.

Second, another future task is to implement the concepts presented in this thesis for other repository solutions (e.g. DSpace, EPrints). As a next step, the definition of a standardized query interface to all repositories would allow to query all available repositories through a centralized query platform.

List of Figures

List of Tables

Bibliography

[1] Tony Hey, Stewart Tansley, and Kristin Tolle, eds. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Redmond, Washington: Microsoft Research, 2009. ISBN: 978-0-9825442-0-4. URL: http://research.microsoft.com/en-us/collaboration/fourthparadigm/ (visited on 10/04/2020).

[2] Monya Baker. "1,500 scientists lift the lid on reproducibility". In: *nature - International weekly journal of science* 533 (2016), pp. 452–454. DOI: 10.1038/533452a.

[3] Colin F. Camerer, Anna Dreber, Eskil Forsell, Teck-Hua Ho, Jürgen Huber, Magnus Johannesson, Michael Kirchler, Johan Almenberg, Adam Altmejd, Taizan Chan, Emma Heikensten, Felix Holzmeister, Taisuke Imai, Siri Isaksson, Gideon Nave, Thomas Pfeiffer, Michael Razen, and Hang Wu. "Evaluating replicability of laboratory experiments in economics". In: *Science* 351.6280 (2016), pp. 1433–1436. ISSN: 0036-8075. DOI: 10.1126/science.aaf0918.

[4] Carmen Maria Reinhart and Kenneth Saul Rogoff. "Growth in a Time of Debt". In: Working Paper Series 15639 (Jan. 2010). DOI: 10.3386/w15639.

[5] Thomas Herndon, Michael Ash, and Robert Pollin. "Does high public debt consistently stifle economic growth? A critique of Reinhart and Rogoff". In: *Cambridge Journal of Economics* 38.2 (Dec. 2013), pp. 257–279. ISSN: 0309-166X. DOI: 10.1093/cje/bet075.

[6] Baker Dean. *How Much Unemployment Was Caused by Reinhart and Rogoff's Arithmetic Mistake?* 2013. URL: http://cepr.net/blogs/beat-the-press/how-much-unemployment-was-caused-by-reinhart-and-rogoffs-arithmetic-mistake (visited on 10/04/2020).

[7] Carmen Maria Reinhart and Kenneth Saul Rogoff. *Reinhart and Rogoff Respond to Criticism*. 2013. URL: https://www.ineteconomics.org/perspectives/blog/reinhart-and-rogoff-respond-to-criticism (visited on 10/04/2020).

[8] Zeinab Zali, Aslanian Ehsan, Mohammad Hossein Manshaei, Massoud Reza Hashemi, and Turletti Thierry. "Peer-Assisted Information-Centric Network (PICN): A Backward Compatible Solution". In: *IEEE Access* 5 (2017), pp. 25005–25020. DOI: 10.1109/ACCESS.2017.2762697.

[9] Dominique G. Roche, Loeske E.B. Kruuk, Robert Lanfear, and Sandra A. Binning. "Public Data Archiving in Ecology and Evolution: How Well Are We Doing?" In: *PLoS Biology* 13.11 (2015), pp. 1–12. ISSN: 15457885. DOI: 10.1371/journal.pbio.1002295.

[10] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J.G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A.C. t Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan Van Der Lei, Erik Van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. "Comment: The FAIR Guiding Principles for scientific data management and stewardship". In: *Scientific Data* 3 (2016). ISSN: 20524463. DOI: 10.1038/sdata.2016.18.

[11] Sue A. Dodd. "Bibliographic references for numeric social science data files: Suggested guidelines". In: *Journal of the American Society for Information Science* 30.2 (1979), pp. 77–82. DOI: 10.1002/asi.4630300203.

[12] Mercè Crosas. "The Evolution of Data Citation: From Principles to Implementation". In: *IASSIST Quarterly* 37.1-4 (2014), p. 62. ISSN: 0739-1137. DOI: 10.29173/iq504.

[13] CODATA-ICSTI Task Group on Data Citation Standards and Practices. "Out of cite, out of mind: the current state of Practice, Policy, and Technology for the citation of data". In: *Data Science Journal* 12 (2013). Ed. by Yvonne Socha. DOI: 10.2481/dsj.OSOM13-043.

[14] Emma Tonkin. "Persistent Identifiers: Considering the Options". In: *Ariadne* 56 (2008). ISSN: 1361-3200. URL: http://www.ariadne.ac.uk/issue/56/tonkin/ (visited on 10/04/2020).

[15] J. Klump and R Huber. "20 Years of Persistent Identifiers – Which Systems are Here to Stay?" In: *Data Science Journal* 16 (2017), p. 9. DOI: 10.5334/dsj-2017-009.

[16] Robert Kahn and Robert Wilensky. "A framework for distributed digital object services". In: *Int. J. on Digital Libraries* 6 (Apr. 2006), pp. 115–123. DOI: 10.1007/s00799-005-0128-x.

[17] Larry Lannom, Lt. Col. Brian P. Boesch, and Sam Sun. *Handle System Overview*. RFC 3650. 2003. DOI: 10.17487/RFC3650.

[18] Andreas Rauber, Ari Asmi, Dieter Van Uytvanck, and Stefan Pröll. "Identification of Reproducible Subsets for Data Citation, Sharing and Re-Use". In: *Bulletin of the IEEE Technical Committe on Digital Libraries* (2016). DOI: 10.5281/zenodo.4048304.

[19] Sarah Jones, Graham Pryor, and Angus Whyte. "How to Develop Research Data Management Services - a guide for HEIs". In: *Digital Curation Centre* (2013), pp. 1–22. URL: http://www.dcc.ac.uk/resources/how-guides (visited on 10/04/2020).

[20] Heinz Pampel, Paul Vierkant, Frank Scholze, Roland Bertelmann, Maxi Kindling, Jens Klump, Hans-Jürgen Goebelbecker, Jens Gundlach, Peter Schirmbacher, and Uwe Dierolf. "Making Research Data Repositories Visible: The re3data.org Registry". In: *PloS one* 8 (Nov. 2013), e78080. DOI: 10.1371/journal.pone.0078080.

[21] Ricardo Carvalho Amorim, João Aguiar Castro, João Rocha da Silva, and Cristina Ribeiro. "A comparison of research data management platforms: architecture, flexible metadata and interoperability". In: *Universal Access in the Information Society* 16.4 (2017). ISSN: 16155297. DOI: 10.1007/s10209-016-0475-y.

[22] P. Louridas. "Version control". In: *IEEE Software* 23.1 (2006), pp. 104–107. ISSN: 1937-4194. DOI: 10.1109/MS.2006.32.

[23] Scott Chacon and Ben Straub. *Pro Git*. 2nd. Berkely, CA, USA: Apress, 2014. ISBN: 1484200772, 9781484200773. DOI: 10.1007/978-1-4842-0076-6.

[24] Spolsky Joel. *Distributed Version Control is here to stay, baby*. https://www.joelonsoftware.com/2010/03/17/distributed-version-control-is-here-to-stay-baby/. 2010. (Visited on 10/04/2020).

[25] Thomson Patrick. *Git vs. Mercurial: Please Relax*. https://importantshock.wordpress.com/2008/08/07/git-vs-mercurial/. 2008. (Visited on 10/04/2029).

[26] Patrick Säuerl. "Data citation under schema evolution in RDBMS". TU Wien. MA thesis. 2018. DOI: 20.500.12708/3504.