

# Data Citation for Evolving Data

## Enhancing the Reproducibility of CSV Research Data Sets

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Wirtschaftsingenieurwesen Informatik**

eingereicht von

**Stefan Pröll**

Matrikelnummer 0217299

an der Fakultät für Informatik  
der Technischen Universität Wien

Betreuung: A.o. univ. Prof. Dr. Dipl.-Ing. Andreas Rauber

Wien, 22. August 2016

---

Stefan Pröll

---

Andreas Rauber



# Data Citation for Evolving Data

## Enhancing the Reproducibility of CSV Research Data Sets

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Business Engineering and Computer Science**

by

**Stefan Pröll**

Registration Number 0217299

to the Faculty of Informatics  
at the Vienna University of Technology

Advisor: A.o. univ. Prof. Dr. Dipl.-Ing. Andreas Rauber

Vienna, 22<sup>nd</sup> August, 2016

---

Stefan Pröll

---

Andreas Rauber



# Erklärung zur Verfassung der Arbeit

Stefan Pröll  
Anichstraße 19 Top 12, 6020 Innsbruck, Austria

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, 22. August 2016

---

Stefan Pröll



# Kurzfassung

Die moderne Forschung ist in immer mehr Disziplinen datengetrieben, denn die Ergebnisse der wissenschaftlichen Arbeit, die traditionell in Journalen publiziert und auf wissenschaftlichen Konferenzen präsentiert werden, basieren auf Forschungsdaten. Wissenschaftler verwenden Daten als Input für ihre Workflows und Experimente und verarbeiten den erzeugten Datenoutput in Prozesspipelines weiter, um ein wissenschaftliches Ergebnis zu erzielen oder nachzuweisen.

Unerlässlich für die wissenschaftliche Integrität ist dabei die Reproduzierbarkeit von Forschungsergebnissen. Fachkollegen müssen die Möglichkeit haben, postulierte Forschungsergebnisse auf deren Stichhaltigkeit und Richtigkeit zu überprüfen. Dazu benötigen Sie Zugriff auf die ursprünglich angewandten Methoden, aber insbesondere auch auf die bearbeiteten Forschungsdaten, die dem Experiment zugrunde liegen.

Die Forschungsarbeit ist ein iterativer Prozess, bei dem Arbeitsabläufe, Methoden und Daten kontinuierlich verbessert werden. Durch die fortwährende Revision von Algorithmen, Tools und Programmen verändern sich die produzierten Daten. Zusätzlich verbessern Wissenschaftler ihre Forschungsdatensets, indem sie Fehler korrigieren, fehlerhafte Daten entfernen oder neue Daten hinzufügen. Aus diesem Grund sind Forschungsdaten nicht statisch, sondern einer kontinuierlichen Evolution unterworfen.

Um in diesem Kontext reproduzierbare Forschungsergebnisse zu ermöglichen, muss die Verfügbarkeit der Daten und daraus abgeleiteter Subsets garantiert werden. Das Gebiet der Data Citation behandelt die Frage, wie Datensets zitierbar gemacht werden können und der Zugriff auf die Daten langfristig gewahrt werden kann. Aus nachvollziehbaren Gründen ist es weder praktisch noch umsetzbar, jedwede Version eines sich kontinuierlich verändernden Datensets individuell abzuspeichern und aufzubewahren. Daher sind neue Methoden notwendig, die über das einfache Archivieren von Daten hinausgehen und den Zugriff auf die Versionshistorie eines Datensets ermöglichen. Die Arbeitsgruppe zum Thema Data Citation (WGDC) der Research Data Alliance (RDA) hat einen vierzehn Punkte umfassenden Leitfaden entwickelt, wie dynamische Daten langfristig zitierbar und zugreifbar gehalten werden können. In dieser Arbeit haben wir den ersten Forschungsprototypen entwickelt, der diesen Leitfaden für CSV Daten implementiert. Mit Hilfe dieser Software können Wissenschaftler reproduzierbare Subsets aus sich dynamisch entwickelnden Datensets erzeugen und auf alle Versionen eines Datensets zugreifen.





# Abstract

Modern research is data driven and the results published in scientific journals and presented at conferences are based upon data. Researchers use data as input and output for their scientific workflows and process and transform during the course of their experiments. Reproducibility is a core requirement for trustworthy science and it ensures that the results can be verified by peers. In order to allow peers to reproduce the research results, scientists require access to the data sets and the methods used in a scientific experiment.

As research is an iterative process, scientific workflows and data are continuously improved. Whenever algorithms advance, tools become adapted or instruments become more precise, the produced datasets change. Also researchers improve their datasets by correcting and purging errors and by including new records into existing datasets. For these reasons research data is not static, but constantly evolving.

In order to enable reproducibility, access to data sets and derived subsets must be guaranteed. The field of data citation deals with the question how data sets can be made citable and accessible for the long term. As it is impractical or even impossible to store each and every subset from evolving data sources, a new method is required which allows to access previously created data sets and subsets. The Research Data Alliance (RDA) Working Group on Data Citation developed a set of guidelines how evolving research data and subsets thereof can be made citable. In this work we present the first research prototype implementation for CSV data, which allows scientists to create citable, precise subsets of evolving CSV data.



# Contents

<b>Kurzfassung</b>	<b>vii</b>
<b>Abstract</b>	<b>ix</b>
<b>Contents</b>	<b>xi</b>
<b>List of Figures</b>	<b>xii</b>
<b>List of Tables</b>	<b>xiv</b>
<b>List of Algorithms</b>	<b>xiv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Research Data and the Scientific Process . . . . .	1
1.2 Motivation . . . . .	2
1.3 Problem Statement and Research Questions . . . . .	4
1.4 Aim of this Work . . . . .	6
1.5 Methodological Approach . . . . .	6
1.6 Structure of the Work . . . . .	7
<b>2 State of the Art</b>	<b>9</b>
2.1 Citing Knowledge - Citing Data . . . . .	9
2.2 Persistent Identifiers . . . . .	13
2.3 Versioning, Databases and Temporal Support . . . . .	17
<b>3 A Data Citation Framework for CSV Files</b>	<b>19</b>
3.1 A Researcher's Workflow for Citable Subsets of CSV Data . . . . .	19
3.2 Making Dynamic Data Citable . . . . .	28
3.3 Summary . . . . .	49
<b>4 Implementing the RDA Recommendations for Data Citation</b>	<b>51</b>
4.1 An Implementation for Citable CSV Data Sets - Overview . . . . .	51
4.2 Supported Workflow . . . . .	70
4.3 Platform Details . . . . .	71
	xi

4.4	Addressing the RDA Recommendations . . . . .	79
<b>5</b>	<b>Evaluation</b>	<b>85</b>
5.1	Overview of the Evaluation Process . . . . .	85
5.2	Results . . . . .	96
5.3	Suggestions and Potential Improvements . . . . .	102
5.4	Application Recommendations . . . . .	104
5.5	Evaluation Summary . . . . .	105
<b>6</b>	<b>Conclusions and Outlook</b>	<b>107</b>
6.1	Summary of the Thesis . . . . .	107
6.2	Future Work . . . . .	108
<b>A</b>	<b>Evaluation Graphs</b>	<b>111</b>
	<b>Bibliography</b>	<b>117</b>

## List of Figures

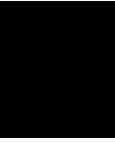
1.1	Versioning by duplication and renaming <sup>1</sup> . . . . .	4
1.2	Reproducibility and repeatability <sup>2</sup> . . . . .	5
2.1	Handle resolver process . . . . .	14
2.2	ARKs locksmith jargon . . . . .	16
2.3	Anatomy of PIDs on the example of ARK . . . . .	17
3.1	CSV sample file . . . . .	20
3.2	File types used in open government data . . . . .	22
3.3	Git diff from two different revisions of a file . . . . .	23
3.4	Git diff after sorting the file . . . . .	24
3.5	Git diff after changing the sequence of columns . . . . .	25
3.6	SQL SELECT statement and the CSV workflow . . . . .	26
3.7	Basic RDA workflow . . . . .	29
3.8	CRUD sequence . . . . .	30
3.9	Versioning records . . . . .	30
3.10	13 time relationships by Allen . . . . .	31
3.11	Computing a hash value . . . . .	38
3.12	Local and global updates . . . . .	39

3.13	PID workflow . . . . .	41
3.14	An example landing page (Dryad) . . . . .	43
3.15	Detailed metadata information in Dublin Core terms . . . . .	44
3.16	DOI information . . . . .	45
3.17	Citation texts for the publication and the data set . . . . .	45
3.18	Statistics page for human actors . . . . .	46
3.19	A Figshare landing page . . . . .	47
3.20	Figshare landing page details . . . . .	48
4.1	ER diagram of the PID system . . . . .	57
4.2	Audit tables of the PID prototype . . . . .	58
4.3	Query Store ER diagram . . . . .	60
4.4	Citation text of the super set . . . . .	64
4.5	Citation text of the subset referencing the super set . . . . .	64
4.6	Example subset with verification information . . . . .	66
4.7	Input of the hash function . . . . .	67
4.8	ER of the user table . . . . .	72
4.9	Main menu . . . . .	73
4.10	Providing the metadata . . . . .	74
4.11	Uploading a file . . . . .	75
4.12	Specifying the primary key . . . . .	76
4.13	A data set landing page . . . . .	77
4.14	Create a subset . . . . .	78
4.15	Success message with the hash and link to the landing page . . . . .	78
4.16	A subset landing page . . . . .	79
4.17	The resolver service interface . . . . .	80
5.1	CSV workflow with Git . . . . .	86
5.2	Git workflow with JDBC . . . . .	87
5.3	Example of a random CSV file . . . . .	89
5.4	Test data set sizes . . . . .	89
5.5	Evaluation database tables . . . . .	92
5.6	Evaluation of the storage demand of SMP-S1 . . . . .	98
5.7	Evaluation of the execution time of SMP-S1 . . . . .	99
5.8	Evaluation SMP-S1-1000 . . . . .	99
5.9	Evaluation SMP-S2 . . . . .	100
5.10	Evaluation SMP-S3 . . . . .	101
5.11	Evaluation SMP-S4 . . . . .	102
A.1	Evaluation MED-S1-100 . . . . .	112
A.2	Evaluation MED-S1-1000 . . . . .	112
A.3	Evaluation LRG-S1 . . . . .	112
A.4	Evaluation MED-S2 . . . . .	113
A.5	Evaluation LRG-S2 . . . . .	113

A.6	Evaluation MED-S3 . . . . .	114
A.7	Evaluation LRG-S3 . . . . .	114
A.8	Evaluation MED-S4 . . . . .	115
A.9	Evaluation LRG-S4 . . . . .	115

## List of Tables

3.1	Data types in open government data . . . . .	21
3.2	Exporting data as CSV . . . . .	26
4.1	Storage requirements in bytes . . . . .	68
4.2	Hash outputs . . . . .	69
5.1	The test data sets . . . . .	89
5.2	The distribution of operations . . . . .	90
5.3	Query distribution . . . . .	91
5.4	Iterations of operations . . . . .	92
5.5	Evaluation overview . . . . .	97
5.6	S1 Evaluation Results . . . . .	97
5.7	S3 Evaluation Results . . . . .	101
5.8	S4 Evaluation Results . . . . .	102
5.9	With and without index comparison . . . . .	104
5.10	Index storage demand . . . . .	104



# Introduction

## 1.1 Research Data and the Scientific Process

Researchers aim to improve the knowledge we have about the world we live in by applying the scientific method. The scientific method is a collection of principles, allowing us to create theories and describe observations made in the real world. Theories are tested against the current state of the art and the current knowledge. Once a theory has been established in a research discipline, it remains the valid until it has been falsified. The scientific method is a principle, which is well established since centuries and it is still accepted across disciplinary boundaries. It is still the best tool that we have for conducting research. The way how science is done in contrast changes and is evolving continuously. The set of tools which constitute a specific way of doing things is called paradigm. For this reason several paradigms have come into existence [35]. Starting with experimental science, where natural phenomena have been explained by experiments, a more theoretical approach was established. Researchers began developing theories utilising equations and developing laws which could describe the phenomena in a more precise way. In the beginning of the 20th century, the development of electronic calculators and their successors, the modern, digital computers allowed to study complex phenomena for the first time, by processing large data sets.

With the technological advancement of computer science, researchers have been able to automate calculations, which used to be a tedious task, and feed data into the machines which could compute results with increasing capacities and speeds. The technical revolution did not only allow the automated computation of equations, but also lead to electronic measurement instruments, being able to produce more and more data which again became difficult to handle with traditional computational machinery.

As a natural consequence, the amount of research data grew, which in turn lead to the Fourth Paradigm [29] called eScience, data exploration or data driven science. Researchers are hardly working with traditional measurement instruments, taking notes and collecting data manually. Data is generated by sensors and other electronic instruments and

scientists process the data by using software on their computers. For this reason the amount of data created is increasing continuously, which introduces new challenges in research methodology in general and computer science in particular.

## 1.2 Motivation

Today we are in the middle of the Fourth Paradigm, where the complete cycle of research is within digital information systems. It has never been easier to create data, which is why the current paradigm is often referred to as the data deluge [28]. Data has become a central aspect of research. Together with algorithms, the processing steps, software and hardware, it is a core pillar of data driven research.

The increase of data can be analysed from several perspectives. Most prominently, the term big data was coined to describe data collections which demand new ways of processing. The term is associated with the Five Vs [21]: volume, velocity, variety, value and veracity. Therefore big data does not only deal with very large databases and data sets, but includes all kinds of data which cannot be processed with standard technologies anymore. Obviously the increasing amount of data in terms of size and complexity and value introduces new challenges to data processing in business, but in particular for the scientific community [8]. As every paradigm shift, also the Fourth Paradigm requires researchers to adapt to the new possibilities and find solutions for the new challenges.

When researchers conduct their experiments, they utilise data sets which are analysed and processed. In many cases the processing leads to result sets, which in turn serve as input for further processing steps. Being able to understand how an experiment was conducted and being able to reproduce the results is a core requirement of the experimental science. Therefore having access to the data and being able to retrieve the correct version of a specific data set is key. For this reason the topic of this thesis is to support researchers during the complete life cycle of their experiment, in managing their data in a reproducible way. The goal is to provide a framework and a concrete tool, which allows the traceable creation of data sets, their refinement and improvement, supports the intermediate and final publication of specific subsets of data and maintains the link between research data and publication for others to reproduce the experiment.

When we started to gather research questions for this thesis, we aimed for Big Data. The topic received a lot of attention and provided more than enough challenges to be tackled. We started to collect data and talked to researchers about their requirements. During the first analysis we realised that in many scientific disciplines the data is not big at all, but still it is difficult to handle valuable data sets and create a lasting link between data, metadata and publications which allows peers to trace the data from the publication to the data set and reproduce the results.

Big Data is dominating the media and gains a lot of attention. This buzzword is often linked to large database clusters or complex data formats. In the scientific domain, the comma separated values (CSV) data format is prevalent in many disciplines. CSV is a very simple and versatile data format which is used for storing and exchanging tabular data. As the format is easy to handle and generate - almost all tools may import or



export - it serves as the most basic data format in many research areas. It does not require specialised storage infrastructure and can easily be edited with standard spreadsheet software. CSV files usually are not huge in terms of file sizes, but as they are easy to store and process, their amount can easily grow too big to handle without proper tool support.

Research is an iterative process and requires to adaptation to new findings. Data sets are updated and adapted constantly during the research workflow to reflect the improvement of an experiment. Therefore data is not static but highly dynamic. Researchers improve their data sets by correcting and purging data and by including new records into existing data sets. For this reason it becomes increasingly difficult for scientists to manage not only the sheer amount of data, but also to keep track of all different versions which may exist. At best scientists use source code management (SCM) tools such as subversion or git. Such systems may only trace differences in the versions of a file but are not suitable for tracing the creation process of a subsets itself. Still researchers need to be able to reliably and efficiently create and cite subsets of large and dynamically growing data sets. Tracing these processes is necessary in order to reference, address, reuse and verify data in the future again and understand how the subset was created in the first place. Therefore a new mechanism is needed which supports researchers in their work with CSV file and provides them with tools for creating citable subsets of their data.

The concept of citing is well established within the scientific realm [40], where researchers publish their findings in peer reviewed papers which are then referenced by peers in their own work. Currently the focus of researchers is still in paper based publications. Data is rather seen as a supplement that could be offered as a download, often without further comments. Yet validation, verification, reproduction and re-usage of existing knowledge can only be facilitated when data is accessible and identifiable. In the context of scientific research data, precise citation mechanisms are more important than ever. The precise identification of a specific subset is necessary for reproducing experiments with exactly the same data basis. The same is true for data sets used within business environments, where the documentation of used data sets serves as an important evidence in liability cases. Therefore data sets need to be identifiable in the future.

There are several reasons for this lack of good data management practices. One of the major reasons is the difference of scientific cultures. Not all scientific disciplines use rigorous data management practices and implement appropriate infrastructure for storing and maintaining their research data. Although the researchers produce high quality data sets which are unique in their field, there is hardly sophisticated database systems, storage devices or other specialised data management software in use. In many cases the data sets are Excel files, text documents or CSV files. Updates to the files are made in place and if version control is utilised, it is often a naive approach based on renaming files. Figure 1.1 shows this approach with a comic.

Introducing specialised systems for handling the data hardly happens. In many cases either the knowledge is not available or the required investments in terms of personnel

---

<sup>1</sup>Source:<http://www.phdcomics.com/comics.php?f=1323>

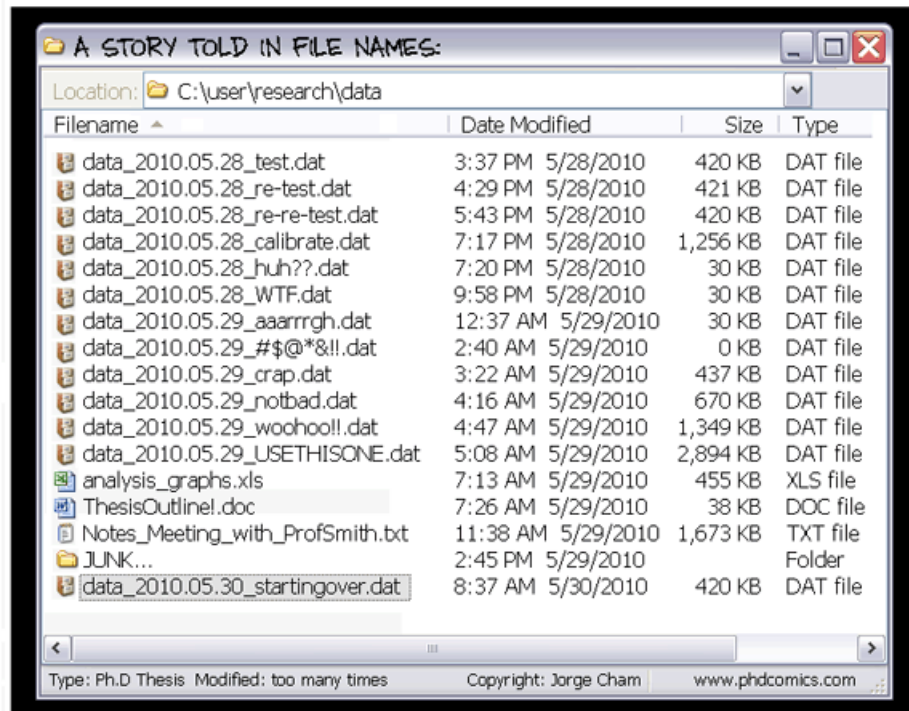


Figure 1.1: Versioning by duplication and renaming<sup>1</sup>

and commitment are not possible. This scenario bears a risk that the data will be lost at some point. This especially true if PhD students finish their degree and move on or if researchers change the institutions. Several cases are known, where researchers stored all their data sets either on their personal devices or on the Web server of their institutions<sup>2</sup>. As soon as the researcher left the organisation, the devices are formatted and inherited to other colleagues and therefore the data is gone. The same is true for data sets stored on Web servers. They are only referenced with URLs which are known to break if not maintained for the long term, which is costly. For this reason many unique research data sets are at constant risk of being lost.

### 1.3 Problem Statement and Research Questions

Two of the major challenges which are introduced by the fourth paradigm is reproducibility and repeatability. Reproducibility deals with the challenge of obtaining the same results,

<sup>2</sup>What would happen if you lost all of your research data? <https://www.digital-science.com/blog/news/what-would-happen-if-you-lost-all-of-your-research-data/>. Retrieved at 18.08.2016

independently of the concrete tools used. Repeatability in contrast aims to re-execute the existing source code and verify the results. Figure 1.2 depicts this relationship.

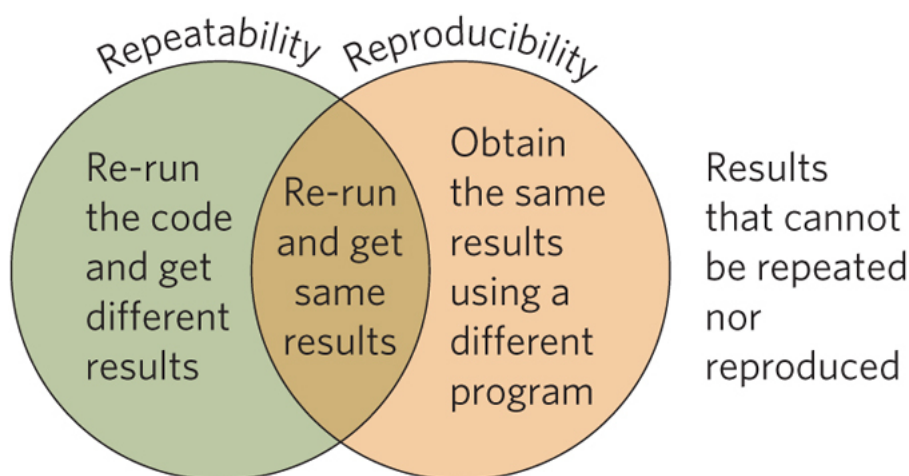


Figure 1.2: Reproducibility and repeatability<sup>3</sup>

Data constitutes a core aspect of reproducible and repeatable eScience experiments. If the research data is not available, the experiment cannot be reproduced and also not repeated. The same is true if several different versions of one data set exists. Researchers need methods which support them in unambiguously referencing the subsets and data sets they want to use in an experiment.

The issues to be tackled in this work circulate around the following questions:

1. What are the requirements for a data citation model supporting evolving CSV data?
2. How can research data sets be versioned and how can subsets be reproduced which have been created from dynamic sources?
3. How can the process of creating subsets be made reproducible?
4. How can researchers verify the correctness of specific versions from subsets of data, without the need for storing individual data exports?
5. How does the system design look like that enables the traceable creation and citation of subsets?
6. What are the steps necessary to migrate existing CSV files into a time stamped relational database scheme to be compatible with the proposed model?
7. What metadata is required for re-producing the subset creation process and for verifying the correctness of a subset?

---

<sup>3</sup>[http://www.nature.com/ngeo/journal/v7/n11/fig\\_tab/ngeo2283\\_F1.html](http://www.nature.com/ngeo/journal/v7/n11/fig_tab/ngeo2283_F1.html)

8. How can the effort for researchers be minimised to produce citable subsets?
9. What properties of data sets need to be considered for enabling long term preservation and fostering access to identical research results?

## 1.4 Aim of this Work

The goal of this work is to answer the research questions from Section 1.3 and to provide a tangible prototype demonstrating the usefulness of the approach developed within this thesis. In order to achieve this goal, this thesis aims to distil the steps necessary for achieving reproducible subsets of data. The sequence of steps, from data acquisition to subsets creation until the verification of the subsets, will be fed into a framework which allows to implement dynamic data citation capabilities. Thus this thesis will provide a theoretical foundation for a framework capable of citing structured research data in a reproducible way. The validity of the framework will be tested by developing a research prototype, implementing the requirements, specified by the framework. The prototype supports the complete life cycle of research data, from data ingestion, processing, subset selection, publication, retrieval and verification.

## 1.5 Methodological Approach

The thesis consist of two parts: a theoretical foundation and a practical implementation. The goal of the theoretical part is to develop a model, capable of referencing subsets of CSV data files based on the recommendations for data citation of the Research Data Alliance (RDA). These recommendations are introduced in Section 2.1.2 and provide guidelines how existing data sources can be made citable.

The practical part will focus on the implementation details of the theoretical part. The model will utilise concepts from temporal databases [32], data versioning [18] and persistent identifiers [26]. As records can change, they need to be versioned, i.e. all changes that affect the data need to be traceable. This entails that no records may be deleted as they are still required for later reference. The same is valid for updates and alterations of records. Hence the proposed solution needs to be able to retrieve specific versions of the data in order to reproduce a data set for its reuse or validation.

The goal of this thesis is to overcome the current principle of only providing subsets of a data set with textual descriptions, without the available metadata to retrieve the same subset again for validation purposes. Therefore the solution needs to provide a mechanism that can address arbitrary subsets of data without the need of storing each individual subset in a separate file structure or copy. The solution will exploit the query features of relational databases and migrate the data from CSV files into a database system. The solution should provide persistent identifiers that allow the unique identification of data sets for the long term and provide all the metadata required for reproducing, verifying data sets.

## 1.6 Structure of the Work

Chapter 1 provides the introduction for this work. It includes the motivation for the topic, the problem statement, the aims of the work and the methodological approach. Chapter 2 provides the state of the art. It covers a generic overview of data citation initiatives and common practices as well as fundamental work in the area of dynamic data citation, persistent identifiers and temporal data. Chapter 3 introduces the data citation framework. In this chapter we describe the research data use case and provide the necessary steps for making data citable. Chapter 4 covers the practical implementation and describes the components developed in detail. Chapter 5 evaluates the solution and compares our implementation with a reference system. Chapter 6 gives a summary and provides an outlook.



## State of the Art

In the following sections we provide an overview of the state of the art in the area of data citation, persistent identification and existing initiatives.

### 2.1 Citing Knowledge - Citing Data

The citation of intellectual work has a century old tradition, standing on the shoulders of giants is one of the fundamental mantras in science. Citing the fundamental work of peers, on which a new finding was based upon, gives credit, establishes a context and makes transparent which knowledge was available before. Giving credit is an important aspect, but not only for valuing the work of peers. More importantly it enables peers to obtain the same prior available knowledge and reproduce the new findings and verify the results. Obviously data has become fundamental for many experiments and studies and therefore requires proper citation.

The evolution of data citation is described by [4], the authors claim that still many research findings do not provide access to the underlying data. Data forms the basis of the results of many research publications, but still there is a large variety in data submission policies and the availability of research data [3, 34].

#### 2.1.1 Current Practice

Publications increasingly contain references to data that was used or generated during the research experiment. Research data sets are often treated as one entity, i.e. indivisible, static and can be referenced as one unit. In many cases, data is referenced bibliographically. As a minimum [9], the following metadata about a data set are required [5]: author, title, date, publisher, identifier and access information. The data itself is then often deposited at an institutional site and referenced by providing an URL. Obviously this mechanism is not suitable for sustainable data citation for several reasons. Uniform Resource Locators (URL) have not been designed to be stable for the long term. As their name implies,

URLs refer to a location, not the object itself. A location may change, for instance when a researcher leaves the institution and her personal Web site is taken down or when a Web server which is hosting the files, is moved to a different domain. As a result, many URLs that served as data citation reference are not accessible any more, a phenomenon known as link rot [42].

### 2.1.2 Initiatives

Several initiatives exist, which deal with data citation and aim to improve standards and awareness for the topic. The following initiatives influence each other and push the development of data citation principles further.

#### **Research Data Alliance (RDA)**

The Research Data Alliance aims to bring together scientists working with research data from various domains. The organisation spans several continents and is organised into interest groups and working groups. The Working Group on Data Citation specifically targets dynamic data citation.

In the case statement, the group identifies four stakeholders, who do have different demands, requirements and views on the topic of dynamic data citation. The benefits and improvements from data citation also differ between the stakeholders:

- Data providers
- Solution providers
- Researchers
- Community

Researchers benefit from data citation by making their data sets and therefore their work visible. Being cited and receiving recognition is an essential incentive in the academic world. Data citation allows to attribute the work from peers and to measure the reuse and therefore acknowledgement of data sets. If scientists make their data available in a citable way, their visibility and credibility increases. Data providers can increase the reuse of their data sets and therefore increase the economic value of their data assets. It allows implementing and utilising new business models by providing citable data sets. Solution providers benefit from data citation by the availability of machine actionable citations and automated services. The scientific community benefits from citable data by increased transparency. Being able to trace and understand where the data came from increases trust and establishes trustworthy research environments.

In contrast to other data citation approaches, which focus on static data sets, this working group views data citation from a more dynamic view. Many scientific settings use some form of data base for storing large data sets. For obtaining specific subsets, query mechanisms can be used, which allow to retrieve only those records, which are of



interest to the researcher. The WGDC relies on query languages and versioned data. Instead of assigning persistent identifiers to static data exports, the WGDC proposes to assign the PIDs to the query instead, allowing to retrieve the appropriate version of a subset from evolving data sources [52, 51].

The WGDC published a set of recommendations, how to render existing data sources citable. This *Recommendations of the Working Group on Data Citation* [55, 56] form the basis of this thesis and the prototype developed in this work is evaluated against the proposed recommendations. More details on these recommendations are given in Section 3.2.

### **Committee on Data for Science and Technology (CODATA)**

The Committee on Data for Science and Technology (CODATA) aims to improve the quality and reliability of research data from several scientific disciplines. Together with the International Council for Scientific and Technical Information, CODATA formed a task group for developing practical and consistent data citation standards. As a result of these efforts, the task group published a report on current data citation practices [46], highlighting the fragmentation of data citation in different scientific areas. In this report, they introduce 10 principles:

1. **Status:** Data citations should be accorded the same importance in the scholarly record as the citation of other objects.
2. **Attribution:** Citations should facilitate giving scholarly credit and legal attribution to all parties responsible for those data.
3. **Persistence:** Citations should be as durable as the cited objects.
4. **Access:** Citations should facilitate access both, the data themselves and associated metadata and documentation. Humans and machines need both to make informed use of the referenced data.
5. **Discovery:** Citations should support the discovery of data and their documentation.
6. **Provenance:** Citations should facilitate the establishment of provenance of data.
7. **Granularity:** Citations should support the finest grained description necessary to identify the data.
8. **Verifyability:** Citations should contain information sufficient to identify the data unambiguously.
9. **Metadata:** Citations should employ widely accepted metadata standards.
10. **Flexibility:** 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 across communities.

## **Future Of Research Communications and E-Scholarship (FORCE11)**

Future Of Research Communications and E-Scholarship<sup>1</sup> is a community aiming to improve knowledge creation and sharing. During a workshop, the so-called Amsterdam Manifesto on Data Citation Principles<sup>2</sup> was developed and published.

1. Data should be considered citable products of research.
2. Such data should be held in persistent public repositories.
3. If a publication is based on data not included in the article, the data should be cited in the publication.
4. A data citation in a publication should resemble a bibliographic citation and be located in the publication's reference list.
5. Such a data citation should include a unique persistent identifier (a DataCite DOI recommended, or other persistent identifiers already in use within the community).
6. The identifier should resolve to a page that either provides direct access to the data or information concerning its accessibility. Ideally, that landing page should be machine-actionable to promote interoperability of the data.
7. If the data are available in different versions, the identifier should provide a method to access the previous or related versions.
8. Data citation should facilitate attribution of credit to all contributors.

## **Data Citation Synthesis Group: The Joint Declaration of Data Citation Principles**

Several groups such as CODATA, FORCE11, RDA and others<sup>3</sup> have been active in developing data citation standards and principles. For harmonising these efforts, the Data Citation Synthesis Group was established. The goal of this group was to find agreement on the basic principles for data citation. As a result, the "Data Citation Synthesis Group Joint Declaration of Data Citation Principles" was published<sup>4</sup>. Institutions are asked to endorse the following eight 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.

---

<sup>1</sup><https://www.force11.org/about>

<sup>2</sup><https://www.force11.org/AmsterdamManifesto>

<sup>3</sup><https://www.force11.org/datacitation/workinggroup> Retrieved at 2015-10-05

<sup>4</sup>Data Citation Synthesis Group: Joint Declaration of Data Citation Principles. Martone M. (ed.) San Diego CA: FORCE11; 2014 <https://www.force11.org/datacitation>.

2. **Credit and Attribution:** Data citations should facilitate giving scholarly credit and normative and legal attribution to all contributors to the data, recognising 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.
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 all 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 verifying that the specific time slice, 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 Persistent Identifiers

To overcome the problem of changing locations, the concept of persistent identifiers was introduced. Persistent identifiers (PIDs) provide unique identification of digital objects and reliable locations of Internet resources. The persistence is based on the maintenance of the links, not a technical aspect of the systems. For this reason the actual location of an object can change, i.e. it can be updated to a new place. This option allows to keep the location correct and refer to the object also if the file system, the domain name or other infrastructure properties change. Thus PIDs require organisational effort for the management for the linking between the data and the identifier.

Also, services for locating and accessing objects are necessary. The organisations providing these services are denoted Registration Authorities (RA). These RAs are responsible for the long term access, resolution and maintenance of the identifiers they issued for digital objects. There exist different solutions for the implementation of persistent identifiers. The authors of [61] provide an overview of the most common approaches and [6] provides implementation details.

### 2.2.1 Handle

The handle System provides the basic infrastructure for identifying digital objects in a persistent way. The Handle System is a protocol, a name space and also the name of the reference implementation [60]. It is a name service which allows resolving identifiers to the location of digital objects. It ensures that the identifiers are globally unique and allows maintaining the persistence of links. The system works in a distributed fashion and at a global scale, hence several instances can provide access to a resource. Equation 2.1 shows the scheme of a Handle identifier. It consists of a prefix, followed by a slash and the resource name, which has to be unique within the prefix.

$$\langle \textit{Handle} \rangle ::= \langle \textit{HandleNamingAuthority} \rangle "/" \langle \textit{HandleLocalName} \rangle \quad (2.1)$$

The prefix is unique and identifies the organisation responsible for the data. This organisation is denoted as the handle naming authority. The resource name is denoted as the handle local name and refers to exactly one resource within the name space of the organisation.

When a user resolves the Handle with the resolution service, the system looks up the prefix or naming authority in the Global Handle Registry and provides the information where to obtain the actual data. This knowledge is then used by the client for obtaining the local handle service which is responsible for the data. Figure<sup>5</sup> 2.1 depicts this process.

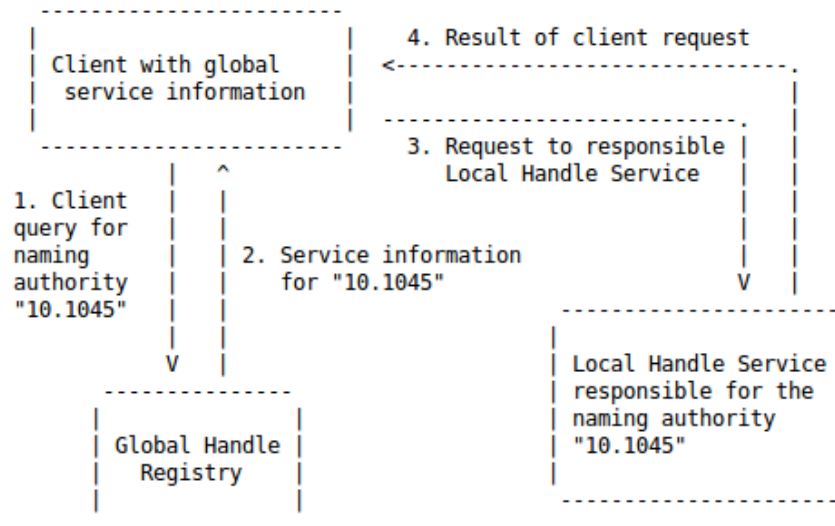


Figure 2.1: Handle resolver process

---

<sup>5</sup>Image source [60]

## 2.2.2 DOI - Digital Object Identifier

The Digital Object Identifier<sup>6</sup> is another popular persistent identifier system. It provides digital identifiers for digital and analogue objects. At the time of writing this thesis<sup>7</sup>, there are more than 114 million DOIs assigned and the annual growth rate is 19 %<sup>8</sup>. It is built on top of the Handle System introduced in Section 2.2.1 and provides additional services, such as a metadata store<sup>9</sup>. DOI uses the Handle System as basic infrastructure for storing and resolving PIDs, as it has been proven to be a robust and scalable system. [Handles by themselves are necessary but not sufficient for the function of the DOI system]<sup>10</sup>, but the DOI ecosystem adds persistence, consistency and semantic interoperability based on extended metadata schemes.

DOI is under the control of the International DOI Foundation and provides actionable and interoperable persistent identifiers. Again the location of the objects and also the associated metadata may be updated whenever needed, but the link between the identifier and this information is maintained. Equation 2.2 shows an example for a DOI, which consists of the central resolver service `http://dx.doi.org/`, the organisational prefix 10.1594 and the suffix *PANGAEA.726855*.

$$\text{http} : // \text{dx.doi.org} / 10.1594 / \text{PANGAEA.726855} \quad (2.2)$$

## 2.2.3 ARK

The Archival Resource Key [37, 38] is a persistent identifier naming scheme developed by the California Digital Library<sup>11</sup>. The system can be used for assigning persistent identifiers to physical, intangible or digital objects. ARK utilises a locksmith jargon, which demonstrates the way how the scheme works as depicted in Figure 2.2.

The system is simplistic and only relies as one central text file containing the currently registered naming authorities. Figure 2.3<sup>12</sup> shows the structure of an ARK PID. In contrast to other schemes, ARK can be hosted on own institutional computers, which is indicated by the replaceable Name Mapping Authority part. ARK can be used for free and they can be deleted, which makes them attractive for testing in the area of software development.

## 2.2.4 Further PID Approaches

There exists a large variety of other PID systems. This availability of different systems is caused by the versatile demands of diverging scientific communities. Examples for such identifier schemes are for instance the International Standard Book Number (ISBN),

---

<sup>6</sup><http://www.doi.org/factsheets/DOIIdentifierSpecs.html>

<sup>7</sup>September 2015

<sup>8</sup><http://www.doi.org/factsheets/DOIKeyFacts.html>

<sup>9</sup><http://www.doi.org/factsheets/DOIHandle.html>

<sup>10</sup><https://www.doi.org/factsheets/DOIHandle.html>

<sup>11</sup><https://wiki.ucop.edu/display/Curation/ARK>

<sup>12</sup>Image source: [http://www.cdlib.org/uc3/naan\\_registry.txt](http://www.cdlib.org/uc3/naan_registry.txt)

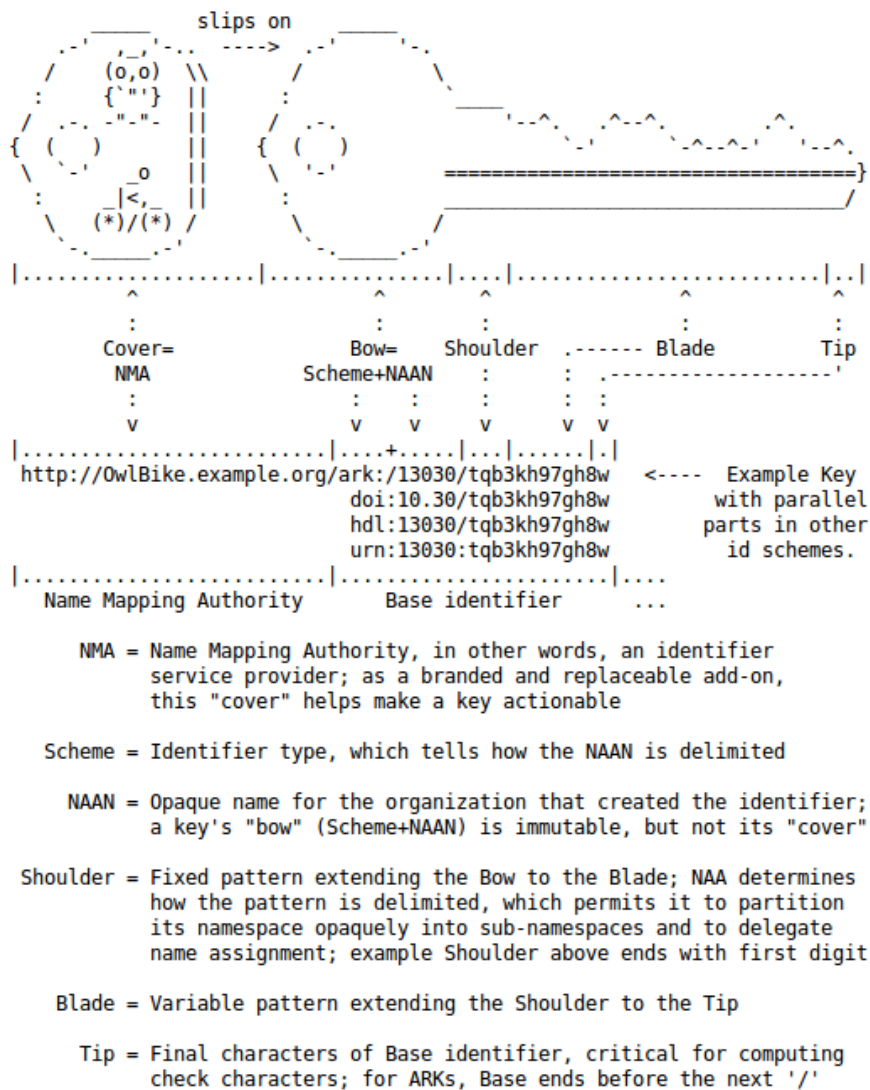


Figure 2.2: ARKs locksmith jargon

International Standard Serial Number (ISSN), International Standard Name Identifier (ISNI), Open Researcher and Contributor ID (ORCID), Persistent Uniform Resource Locator (PURL) and many other concepts based on Uniform Resource Identifier (URI).

### 2.2.5 Shortcomings of Persistent Identifier Systems

Although persistent identifiers solve the problem with locations of digital objects, there are drawbacks for dynamic data. As stated in the enumeration above, the granularity of the identifiers can be adjusted to the requirements of the data set. Subsets require their own identification and metadata. Assigning persistent identifiers (PIDs) to data portions

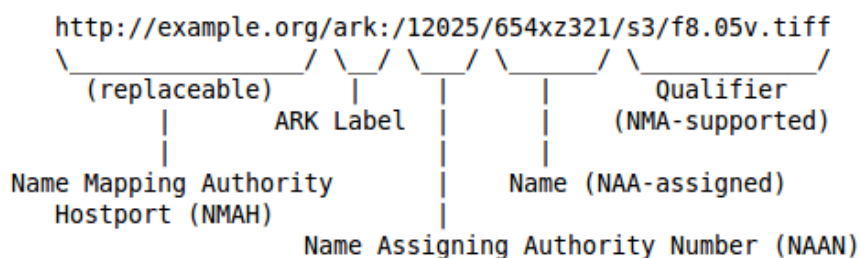


Figure 2.3: Anatomy of PIDs on the example of ARK

of finer granularity, i.e. CSV file rows or even cells would require enormous numbers of unique identifiers and yield unfeasible citations. PID approaches are suited very well for static data, which should only serve as reference point once it has been created. Using the identification and additional metadata is sufficient to search, identify, and retrieve data again. However, many settings require us to go beyond these limitations and introduce scalable and machine-actionable methods that can be used in dynamically changing, very large data sets or high amounts of data files. Also, many data sets continue to grow and are updated as the data sets are used in experiments. In order to enable data citation in dynamic environments versioning support is required. Furthermore, different stakeholders may be interested in diverse portions of the data. Hence, clearly defined subsets of the data need to be identifiable and citable as well. These are some reasons why PIDs assigned to entire data sets or databases are not sufficient for several applications.

The services also differ in the complexity of the registration process. Integrating existing persistent identifier systems such as Handle or DOI requires the registration of a prefix<sup>13</sup> and the payment of licence or service fees<sup>14</sup>.

The work described in this thesis strongly uses the concept of persistent identifiers. Although different systems exist, we decided to keep our implementation independent from any suggested service and implemented a own PID system which is suitable for the prototypical scenario of this thesis. One goal of this work was to provide a data citation service generic enough to be used in several different communities. For this reason we prefer a solution agnostic of the PID system used.

## 2.3 Versioning, Databases and Temporal Support

Databases are frequently involved in business and scientific process execution, where they are used for storing large amounts of data. Database management systems (DBMS) provide capabilities such as isolated transactions, referential integrity and efficient access structures. The ACID requirements for transactions [25] include atomicity, consistency, integrity and durability and are implemented by many DBMS. These features allow

<sup>13</sup><http://www.handle.net/prefix.html>

<sup>14</sup><https://www.doi.org/faq.html>

storing complex data in a reliable way and hide the complexity via transparent interfaces from the user.

Many different types of DBMS exist, which are based upon various database models. The most commonly used database model is the relational database model [16], which is implemented by relational database management systems (RDBMS) such as Oracle MySQL, PostgreSQL, Microsoft SQL Server, Oracle, IBM DB2 and many more. The relational database model allows to map aspects of the real world to relational tables and link these tables with each other. Due to its flexibility, the relational model is successfully used in science and commerce for decades [17].

Query languages such as SQL [19] can be used for interacting with the database management system and retrieving precisely defined subsets of data. SQL is a declarative language which allows retrieving subsets of relational data based on the selection of columns and the projection of rows. The results delivered by a SQL query is often the basis for further processing in scientific experiments.

In many scenarios the data is not static but frequently updated. To understand which data was used in an experiment, comprehensive versioning and temporal query are required. Database management systems can be used for storing temporal data, by versioning the data records and utilising timestamps to store each update, insert or delete of a record [11, 62, 15]. Using the concept of time has a long tradition in database research [39], capturing the sequence of events is essential [2]. Temporal databases [59] implement the bitemporal data model [31] which is based upon valid time and transaction time. Valid time describes the period when a fact was true in the real world, whereas transaction time captures the time frame between the insertion and the deletion of a record in the database system [32].

Most database management systems provide capabilities for storing temporal information [12] and versioned data [30]. The recently updated ISO SQL:2011 standard includes temporal features such as system versioned tables [36]. More and more commercial RDBMS such as MS SQL Server<sup>15</sup>, Oracle 10g<sup>16</sup> or DB2 10<sup>17</sup> support temporal data natively and allow retrieving previously existing data. These technologies add temporal metadata columns and provide keywords for querying temporal data.

Recently also version control systems such as Git are getting increasingly popular for enhancing the reproducibility of experiments [54, 20]. Traditionally these systems have been used for tracing source code in collaborative software projects, but recently also scientists from other domains start using source code management systems for managing their research data and scripts [23].

---

<sup>15</sup>Microsoft SQL Server 2016 (<https://msdn.microsoft.com/en-us/library/dn935015.aspx>)

<sup>16</sup>Oracle 10g ([http://docs.oracle.com/cd/B19306\\_01/server.102/b14200/statements\\_10002.htm](http://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_10002.htm))

<sup>17</sup>IBM DB2 10 (<http://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/>)



# A Data Citation Framework for CSV Files

## 3.1 A Researcher's Workflow for Citable Subsets of CSV Data

Currently a lot of research is still paper driven. This means that researchers aim to publish their results in prestigious papers and journals. In many cases the results are based on data. Researchers store their data in a broad range of different file formats. Some file formats such as CSV are very generic and used in many different domains, whereas others such as for instance the NASA Ames Format<sup>1</sup> are only used in specific domains. What most data formats have in common is that they offer the possibilities to create subsets from a larger data source. By creating subsets, researchers can focus only on those aspects of a potentially large data set in which they are interested in.

The generic research data workflow begins with the collection or creation of a data set. Data is generated by software applications, instruments or sensors during the execution of a scientific workflow. It is a common goal to make research data available and share it with the peers of a specific domain. In many scientific applications only a specific view on the data is required in order to conduct research. Hence researchers do not use the complete data set. Instead they create a specific subset by selecting, filtering and sorting the data set in order to retrieve a subset. This sub-setting process is intellectual work, it requires profound knowledge of the domain in order to make informed decisions about what to include into a data set and which records to omit.

In terms of reproducibility, this process currently is not captured and therefore a lot of knowledge is lost. In some cases at least the subset is documented informally and allows with considerable effort to re-create a specific data set. In other cases the data might be inevitable lost and the creation of a subset cannot be reproduced. As a consequence,

---

<sup>1</sup><https://badc.nerc.ac.uk/help/formats/NASA-Ames/>

the result cannot be reproduced by peers and it is not possible to verify the experiment. Storing all subsets is not a feasible approach as the amount, granularity and complexity of data subsets varies. Research data is constantly improved and updated in an iterative fashion. As described in the motivation in Section 1.2, storing and maintaining all versions of highly specific subsets does not scale.

The RDA Working Group for Data Citation provides a generic framework for data citation, which contains guidelines how existing dynamic data sources can be made citable. We utilise a real world use case for the development and refinement of the RDA data citation framework targeting CSV files. By choosing this most generic data format as use case, we can focus on the key aspects of data citation without having to deal with the peculiarities of a specific data format.

In the following sections we introduce a workflow for CSV data and demonstrate how the researchers can create citable and verifiable subsets from evolving CSV data.

### 3.1.1 Comma Separated Values Files

The comma separated values file format is a simple file format which allows storing tabular data in columns and rows. CSV is a plain text format and is for this reason human readable with any editor. The file format stems from the area of spreadsheet applications and has never been formally standardised [57]. Recently there have been efforts of standardising CSV, for instance by providing a schema for CSV files<sup>2</sup> or by establishing a Web Working Group for CSV Data<sup>3</sup>.

It has become a legacy and data exchange format for many applications as it is easy to parse and write. A CSV file contains sequences of characters in ASCII. Every record within a CSV file occupies one line which is delimited with a carriage return or line break. The last record (or line) of a CSV file does not need to contain a line break. One record can contain several fields, which are delimited by a comma, thus the name CSV. Each record must contain the same number of fields, similar to a matrix which has the same dimensions for each row. Each of the fields of a record can optionally be enclosed within double quotes. If a double quote should be part of the string stored within a field, they must be escaped with another double quote. A CSV file may contain a header which provides the field names. The header record is structured the same way as other records. Figure 3.1 show a CSV data excerpt from the Million Song Data Set [7].

```
song_id,title,artist_name,duration,year
S0FSQUC12A58A7DE35,If You Need Me,The Rolling Stones,122.93179,1964
SORUZGC12A6D4F71F5,Having A Party,Toots & The Maytals,171.98975,1974
S0TAYUF12A8BED793C,Something's Changed,Sharon Jones & The Dap-Kings,176.87465,2007
S0UIHHP12D0219A2F0,Cantaloupe Island,Herbie Hancock,325.04118,1964
S0PDLBH12AB0188BD5,Hypocrite,Antibalas,306.1024,2002
```

Figure 3.1: CSV sample file

---

<sup>2</sup><http://digital-preservation.github.io/csv-schema/csv-schema-1.0.html>

<sup>3</sup>[http://www.w3.org/2013/csvw/wiki/Main\\_Page](http://www.w3.org/2013/csvw/wiki/Main_Page)

As CSV is not standardised, different variants exist. For this reason, CSV files often utilise other delimiters, such as semicolons, pipes or tabs. The escaping of the values is either omitted or realised by using either single or double quotes.

Nevertheless the format remains easy to parse and read and is therefore an ideal candidate for exporting and importing back the data.

### 3.1.2 CSV in the Wild

Providing access to the data allows reusing existing resources in new contexts. Just providing access to the data does not guarantee the usability thereof<sup>4</sup>, as often specialised tools are required to open and read proprietary formats. Formats where the specification is published and where public or open source implementations of editors and readers exist are denoted open formats. Recent open data initiatives such as in the UK<sup>5</sup>, USA<sup>6</sup> or Austria<sup>7</sup> provide access to government data for the public. Most of these portals offer a range of formats for their data sets and the majority of the formats are in plain text, allowing simple processing and human readability. As demonstrated in Table 3.1, CSV is still a widely used format. More than 50 % of the data sets from the open data portals of the UK and Austria for instance are available in CSV, as depicted in Figure 3.2<sup>8</sup>.

File types	UK	Austria
<b>CSV</b>	<b>4692</b>	<b>996</b>
HTML	2288	172
JSON	440	210
PDF	1085	144
RDF	278	0
XML	703	139

Table 3.1: Data types in open government data

Because of the prevalence of CSV in available data sets, we decided to focus on this data type in this work. One of the reasons why CSV is still very popular is based on its simplicity, elegance and functional availability in many applications. Not only is CSV supported by spreadsheet software such as Excel, but also many database management systems, such as MySQL or H2 provide export and import functions from relational data structures into CSV. Thus CSV often also serves as a backup format for relational database management systems (RDBMS) and it is used for migrating data between systems, create and export data extracts or subsets, which are then processed independently.

---

<sup>4</sup><http://www.w3.org/DesignIssues/LinkedData.html>

<sup>5</sup><http://data.gov.uk>

<sup>6</sup><http://www.data.gov>

<sup>7</sup><https://www.data.gv.at>

<sup>8</sup>Data collected from the portals at 06.07.2015

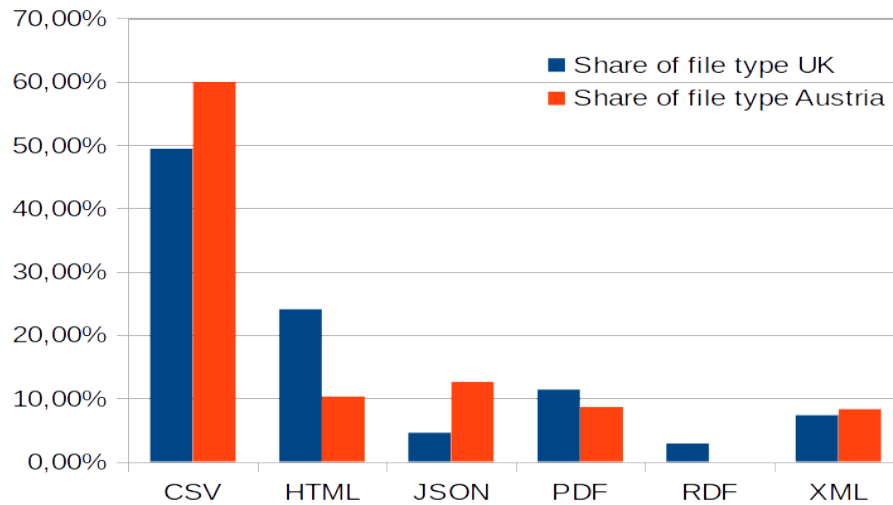


Figure 3.2: File types used in open government data

### 3.1.3 The Current Approach of CSV in Research

Researchers often use CSV data in scientific experiments and for their calculations in spread sheet and statistical software, for reports and for visualisations. In many cases the research data sets are created by exporting a subset from a larger data source, such as a relational database management system (RDBMS). Creating an individual subset allows researchers to focus on a specific problem and reduce the complexity by removing unnecessary data. As research is a dynamic process, the creation of subsets is iterative. Thus the underlying data source may change, errors can be detected and corrected and new data could be included into the data set. Thus the researcher needs to re-create the subset after each update of the data source, which is a cumbersome task. The way how the data was changed is often not recorded and no provenance data of the process exists. This hinders reproducibility of scientific experiments. Also the subset creation process itself carries valuable metadata about how a data set was created and what decisions influenced the inclusion or exclusion of a record into a subset.

The generic workflow of creating a subset is as follows: First, the researcher decides which CSV file to use. Depending on the purpose and properties of the experiment, modifications such as adding, deleting or modifying data may be necessary. In a third step, the actual subset of data is created, by limiting the amount of columns and only including those which carry values the researcher is interested in. The data set can be refined further by filtering the records or rows respectively of a data set. Records (rows) of the subset can be included or excluded based on a set of criteria, such as data ranges, wild card filters or regular expressions. Additionally, the data can be sorted by columns and also the sequence of the columns within the file can be specified. As a result, the researcher receives an individual subset based on specific properties tailored for the purpose of an experiment. It is obvious that these data set properties need to be preserved for being able to reproduce an experiment. This includes the whole process of

subset creation and the data itself.

### 3.1.4 Versioning and Copies

Managing an evolving data set over a longer period of time is a cumbersome task. As already highlighted in the motivation in Section 1.2, researchers either only keep the latest version of a data set and do not store previous versions appropriately. If they are interested in previous iterations of a data set, the researchers often utilise a rename and copy approach, where each version of a data set is distinguished by its file name. The naming of data files does not follow a defined scheme and the file names are often ambiguous. The metadata stored by the file system, such as most recent access or change dates, can easily be altered or manipulated and do not provide a reliable provenance trail.

Recently source code management software and distributed revision control systems such as Git<sup>9</sup> or Subversion<sup>10</sup> is spreading from the software development departments to the labs, as the software allows working collaboratively on files and trace changes. These systems have been designed for plain text file formats, as their change detection algorithms are based on the comparison of strings. If the version control system (VCS) is used consequently and each change of the file is committed into the source code repository, the changes are traceable and previous versions of each file can be compared with the current revision. Figure 3.3 shows an example of a so called diff. The git-diff command shows the differences between two files or two different revisions of files. One single field value of a CSV file was changed, leading one completely updated line in the file. Git shows this difference as a delete with a following add of the complete line.

```
diff --git a/mozart.csv b/mozart.csv
index 1da3ca9..25543d9 100644
--- a/mozart.csv
+++ b/mozart.csv
@@ -1,5 +1,5 @@
 song_id,title,release,artist_name,duration
-SOABXSF12A8C1389B2,Ecco Il Birbo Che Tha Offesa (Don Giovanni_ Act 1,Don Giovanni,Wolfgang Amadeus Mozart,350123
+SOABXSF12A8C1389B2,Ecco Il Birbo Che Tha Offesa (Don Giovanni_ Act 1,Don Giovanni,Wolfgang Amadeus Mozart,313.52118
SOYVRJN12A8C144F71,Utopia [Feat Chris Jones],Utopia,Within Temptation,236.52223
SOMALFU12AB017E1A4,Lügner,Steh grad,Wolfgang Ambros,211.40853
SOLZHTF12A8C136AB9,Monday Monday,California,Wilson Phillips,196.8322
```

Figure 3.3: Git diff from two different revisions of a file

Git also provides metadata about each change which includes timestamps of when the change was introduced, the name and path of the updated files and the index hash. Additional metadata can be retrieved from the log maintained by Git. This includes information about the author who committed a change to the Git repository, the date and a commit message. Listing 3.1 shows a simple Git log.

Listing 3.1: Git log excerpt

```
1 commit 6d8c8fa46a7ce2d4991e5f967411cad1a2b0e06b
2 Author: stefanproell <info@stefanproell.at>
```

<sup>9</sup><https://git-scm.com/>

<sup>10</sup><http://subversion.apache.org/>

```

3 Date: Fri Aug 19 17:28:13 2016 +0200
4
5     Sorting songs by their duration
6
7 commit blf4f479f030feb7bf39af3b569a62b7e74943f8
8 Author: stefanproell <info@stefanproell.at>
9 Date: Fri Aug 19 17:27:39 2016 +0200
10
11     Swapping columns
12
13 commit f32fad6be3b8e653dc18ee77adf7a2615ed5b3bd
14 Author: stefanproell <info@stefanproell.at>
15 Date: Fri Aug 19 17:26:02 2016 +0200
16
17     Initial import
18
19     init

```

While this information is sufficient for detecting and tracing changes, it does not provide methods for tracing the other properties of data sets introduced by the scientific workflow. Sorting a column or re-arranging the sequence of columns of a CSV file is reflected on line level only, thus the traceability is limited because changes below row level granularity are not detected. Sorting for instance can hardly be differentiated from updating records, which also involves the deletion and subsequent addition of a record into the file. Figure 3.4 depicts the diff for the CSV file after sorting it by the *artist\_name*.

Re-arranging the columns of a CSV file by changing their sequence also leads to a completely different file according to the log, as all of the records are considered as deleted and new records are detected to be added. Figure 3.5 shows this example, where the columns *artist\_name* and *song\_id* have been swapped.

```

diff --git a/mozart.csv b/mozart.csv
index 1da3ca9..ae8533c 100644
--- a/mozart.csv
+++ b/mozart.csv
@@ -1,8 +1,8 @@
 song_id,title,release,artist_name,duration
+S0CW0Z12A8C137973,Bronze Silber und Gold,30 Jahre,Wolfgang Petry,193.51465
+S0MALFU12AB017E1A4,Lügner,Steh grad,Wolfgang Ambros,211.40853
+SOABXSF12A8C1389B2,Ecco il Birbo Che Tha Offesa (Don Giovanni Act 1,Don Giovanni,Wolfgang Amadeus Mozart,350123
+S0IIMTS12AB018806B,Serenade In Strings,Classical Selections (Digitally Remastered),Wolfgang Amadeus Mozart,113.68444
+S0YVRJN12A8C144F71,Utopia [Feat Chris Jones],Utopia,Within Temptation,236.52223
-S0MALFU12AB017E1A4,Lügner,Steh grad,Wolfgang Ambros,211.40853
-S0LZHTF12A8C136AB9,Monday Monday,California,Wilson Phillips,196.8322
-S0CW0Z12A8C137973,Bronze Silber und Gold,30 Jahre,Wolfgang Petry,193.51465
-S0LVBQ012AF72A2299,It's Too Late,It's To late,Wilson Pickett,192.86159
-S0IIMTS12AB018806B,Serenade In Strings,Classical Selections (Digitally Remastered),Wolfgang Amadeus Mozart,113.68444
+S0LZHTF12A8C136AB9,Monday Monday,California,Wilson Phillips,196.8322

```

Figure 3.4: Git diff after sorting the file

Creating subsets from a given CSV file corresponds to applying filtering (deleting), sorting and changing the sequence of the columns of the CSV data and therefore creates similar traces in the log files which makes tracing the changes more difficult. It is important to stress that this is obviously not a flaw in Git, but rather a demonstration that the tools does not fit for the purpose of creating reproducible data sets from evolving data sources.

We implemented an approach for data citation based on Git. Details on this approach will be given in Section 5.1

```

diff --git a/mozart.csv b/mozart.csv
index 2110818..38c6e6a 100644
--- a/mozart.csv
+++ b/mozart.csv
@@ -1,8 +1,8 @@
- title,song_id,release,artist name,duration
-Monday Monday,SOLZHTF12A8C136AB9,California,Wilson Phillips,196.8322
-Ecco Il Birbo Che Tha Offesa (Don Giovanni Act 1,SOABXS12A8C1389B2,Don Giovanni,Wolfgang Amadeus Mozart,313.52118
-Utopia [Feat Chris Jones],SOYVRJN12A8C144F71,Utopia,Within Temptation,236.52223
-Lügner,SOMALFU12AB017E1A4,Steh grod,Wolfgang Ambros,211.40853
-Bronze Silber und Gold,SOCWXOZ12A8C137973,30 Jahre,Wolfgang Petry,193.51465
-It's Too Late,SOLVBQQ12AF72A2299,It's To late,Wilson Pickett,192.86159
-Serenade In Strings,SOIIMTS12AB0188D6B,Classical Selections (Digitally Remastered),Wolfgang Amadeus Mozart,113.68444
+song_id,title,release,artist_name,duration
+SOLZHTF12A8C136AB9,Monday Monday,California,Wilson Phillips,196.8322
+SOABXS12A8C1389B2,Ecco Il Birbo Che Tha Offesa (Don Giovanni Act 1,Don Giovanni,Wolfgang Amadeus Mozart,313.52118
+SOYVRJN12A8C144F71,Utopia [Feat Chris Jones],Utopia,Within Temptation,236.52223
+SOMALFU12AB017E1A4,Lügner,Steh grod,Wolfgang Ambros,211.40853
+SOCWXOZ12A8C137973,Bronze Silber und Gold,30 Jahre,Wolfgang Petry,193.51465
+SOLVBQQ12AF72A2299,It's Too Late,It's To late,Wilson Pickett,192.86159
+SOIIMTS12AB0188D6B,Serenade In Strings,Classical Selections (Digitally Remastered),Wolfgang Amadeus Mozart,113.68444

```

Figure 3.5: Git diff after changing the sequence of columns

### 3.1.5 A Query Centric approach: The Query Store

So far researchers have mainly been using simple versioning approaches as described in Section 3.1.4. The duplication of data adds complexity to the preservation of the data and the available metadata is not sufficient for describing subsets properly. Several versions of data sets and data files need to be traced and the unique identification of several versions of a file requires stringent data management. Source code management systems can mitigate the versioning problem, but they do not support the subset process and currently no mechanisms exist, which allow to preserve the creation process of a subset from an evolving data source.

With increasing data set sizes, storing data sets in single files also becomes cumbersome. Creating subsets from CSV files for instance, is often done with spreadsheet software, but naturally these applications have limits with respect to the amount of columns and rows which can be handled. The steps performed by a user to create a subsets are not traceable neither, often whole tables are copied and merged with no additional metadata available. Also the quality of data management is improving and the infrastructure for storing the data is getting more sophisticated. Even though a lot of data is available in the CSV file format, there is a tendency towards more advanced formats and storage systems.

#### Using Databases for Retrieving CSV

For larger data sets, databases have proven to be a reliable choice for storing, processing and retrieving data. CSV still serves as a popular export format for database systems, as it allows the exchange of data with applications which do not yet have database connectivity.

Databases allow storing very large volumes of data and producing specific subsets of a data source on demand, by using descriptive query languages. Database query languages such as SQL allow creating specific subsets of large data sources, including filtered records and defining the sorting of records within a column and the sequence of the columns in a

Database System	Export result as CSV	Dump table as CSV
MySQL <sup>11</sup>	SELECT, GUI	SELECT, GUI, CMD
PostgreSQL <sup>12</sup>	CMD, SP	SP
SQL Server	SELECT, CMD, GUI	SELECT, CMD, GUI
H2 Database <sup>13</sup>	SP	SP

Table 3.2: Exporting data as CSV

data set. Relational database management systems (RDBMS) use sets as their internal structures and they allow to store data and present result as tables. For this reason they are very applicable to CSV files as they share common concepts such as records (tuples), rows, columns. Also the sub-setting process is very similar in both approaches. The basic methods needed for creating subsets are filtering and sorting. Figure 3.6 show the CSV sub-setting process in comparison to the equivalent SQL statement.

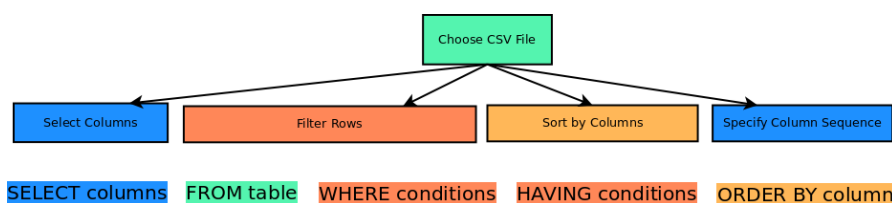


Figure 3.6: SQL SELECT statement and the CSV workflow

For this reason query languages can be used for creating precisely defined subsets of data sources. Instead of manually creating a subset by filtering and sorting a CSV file, researchers can also use databases such as RDBMS in order to retrieve the data sets in CSV file format. Thus the data does not necessarily have to be available natively in CSV, but can also be persisted in a database and retrieved on demand. Table 3.2 shows examples how data can be exported from various RDBMSs by using *SELECT*-Statements or stored procedures (*SP*) or command line tools (*CMD*) or graphical user interfaces (*GUI*).

### The Advantages for Storing CSV Data in RDBMSs

Naturally CSV data can also be imported into RDBMS systems and the data can be stored in database tables allowing efficient access to data. Relational database management systems enable reliable, multi user access to data on a much more fine granular level as generic file systems allow. Standard CSV files also do not have a schema attached, which entails that the column type is not defined. Although there is no limit for the file size

<sup>11</sup><https://dev.mysql.com/doc/refman/5.7/en/select-into.html>

<sup>12</sup><http://www.postgresql.org/docs/current/static/sql-copy.html>

<sup>13</sup><http://www.h2database.com/html/functions.html#csvwrite>



besides the limits of the file system, very large CSV files become difficult to search and handle as they do not provide efficient file access structures. The advantage of database management systems is evident, as indices allow efficient search and access.

The biggest benefit for using RDBMS obviously lies in the fact that they provide a query language. As depicted already in Figure 3.6, the SQL query necessary for creating a subset of CSV data involves a specification of the columns to be included and the criteria for selecting which records should be included into the subset. Finally, each subset can be sorted by considering several columns in ascending or descending order. As the queries are plain text, they can be easily stored for later reference. The structure of CSV files can be mapped to single tables, which renders the query complexity simple, as in this work we only consider single table SELECT statements with non-aggregate columns. Therefore all necessary information for describing a subsets can be expressed with a SQL SELECT statement. Thus if the desired column sequences are defined and the conditionals for including or excluding records are known, the subset is not only described to a large extent, but also the query and therefore the subset are reproducible.

The queries (i.e. SELECT statements) can either be persisted as plain text or their parameters such as table name, selected columns, applied filters and limits as well as the sorting and the sequence of the columns can be stored.

Database systems not only allow efficient data querying, but they also provide integrated metadata collection and management. Storing additional information such as timestamps, user data and further information can either be achieved by adding further columns into each table or by storing the metadata in a dedicated table.

## Databases as Repositories

Storing data as flat files without additional metadata, data protection and preservation information, brings research subsets at risk. For this reason recently data repositories are used for professionalising data storage, access and processing. Repositories use defined procedures and methods for managing the data and therefore they increase the reliability of data storage and access. Frameworks such as the Reference Model for an Open Archival Information System [14] (OAIS) provide guidelines how to design repositories. The OAIS describes how data producers, consumers and management actors interact with each other within the repository. The reference model describes how data provided by the producers can be imported (ingested) into the repository, how it can be managed, administrated, preserved and disseminated to users.

If the data is generated with external applications, usually the data can be exported as CSV files. These CSV files can then be imported into the RDBMS, by creating a table structure based on the available columns and import the data in a second step. After the data has been imported, the query mechanism can be utilised for creating the subsets. The results of the SQL queries can be exported as CSV again and fed back into the scientific process which utilises the data as input. So far this approach helps to conveniently create subsets of data, but additional components are needed for being able to deal with evolving data and for storing the metadata appropriately.

Our approach utilises the query mechanism of relational databases which allows retrieving subsets of specific versions of evolving data. We store the query, its execution parameters as well as the time when the query was executed. We can then re-execute the query on the data as it was at a specific point in time and retrieve the data again. By focusing on the query instead of the data export, we can implement a lightweight citation mechanism which only requires to preserve the query and its metadata in addition to the versioned data source. The following sections we describe the process of making existing data sources citable.

## 3.2 Making Dynamic Data Citable

In the previous section we described different approaches for creating subsets from CSV file and provided current workflows how scientists work with their data. In the following sections, we describe the approach suggested by the Research Data Alliance Working Group on Data Citation (cf. Section 2.1.2). In the following, we provide detailed steps how existing data can be made citable while still allowing changes and updates to the data source based on these recommendations. We describe how each of the recommendations relates to the CSV use case in this section and provide implementation details how we addressed each recommendation in Chapter 4.

### 3.2.1 The Recommendations for Data Citation

The RDA WGDC structures the citation process into three sub-processes and adds a long term perspective for managing change in the infrastructure. The sub-processes are depicted in Figure 3.7 and described in more detail in the following sections. The preparation process is required for preparing the data storage and the so called query store. This first step is the first phase for rendering existing data citable. The preparation of the data store only needs to be done once, when an existing data source should be rendered citable. Technology changes might require to prepare the newer data storage as well. Details about the preparation phase are given in Section 3.2.2. The second sub-process deals with identifying specific subsets and enabling long term access to the data sets. It describes how the queries which are issued at a specific timestamp in the data store can be persisted and utilised for retrieving the very same data set again at a later point in time. This process step is also responsible for attaching persistent identifiers to the subsets. Details about the subset persistence are described in Section 3.2.3. The third phase is denoted the retrieval phase, which describes the steps necessary to retrieving a subset again. More information about the last sub-process is provided in Section 3.2.4. All three sub processes are accompanied from a long term perspective, which is denoted as technology and infrastructure change. As we are considering long term data access to the subsets, technological advancement has to be considered. This is described in Section 3.2.5.

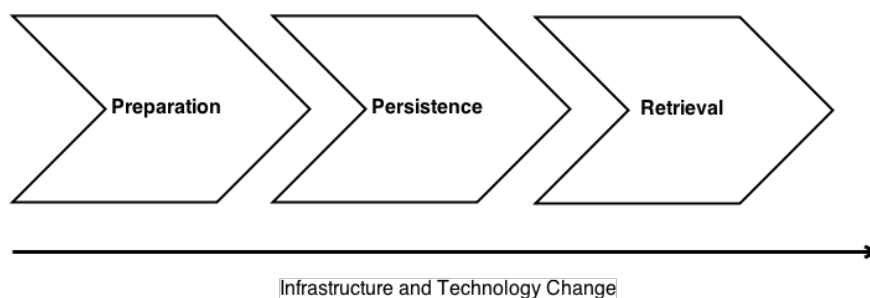


Figure 3.7: Basic RDA workflow

### 3.2.2 The Preparation Sub-process

The first of the three RDA dynamic data citation sub-processes deals with the preparation of the data store and the query store. As the approach is based on versioned data and timestamped queries, the data storage system needs to support versioning. In most database applications, this is a common task and often already implemented. Additional details on versioning are given in Section 3.2.2. Timestamping is required for creating a consistent mapping between the versioned records and the query execution time. Information about the timestamping is given in Section 3.2.2 For the query based mechanism, the query store needs to be prepared, which is used for persisting the queries for the long term. The Query Store basics are described in Section 3.2.2.

#### R1 - Data Versioning

A core principle of reproducible data sets is their versioning. In the scientific realm the data is evolving constantly, which entails that we need to store all previous versions of a data set in order to have access to earlier states of the data set. The dynamic data citation approach utilised data which is stored in database systems, thus the database management system needs to store the records in a versioned fashion.

When dealing with research data, the classical CRUD events are considered: create  $C$ , read  $R$ , update  $U$  and delete  $D$ . Figure 3.8 shows the life cycle of data in databases.

The events which change the data in the database need to be captured and their effects on the records need to be stored in the database. Therefore every insertion, update or deletion of data requires that this operation is stored together with the timestamp of its occurrence. Details on timestamps are given in Section 3.2.2 and Figure 3.9 shows an example for a simple versioning scheme.

The simplest versioning scheme duplicates each record after each change and stores the new value together with the event type and the timestamp of its occurrence. In the example from Figure 3.9 the granularity of the time is one day and each event is stored with the full record attributes. By storing each record with its timestamp, the state of a data set can be reconstructed by only retrieving those records which have been valid during the execution time. A record is considered valid if it has been created before the query execution time and has either an associated  $C$  or  $U$  event active. Records which

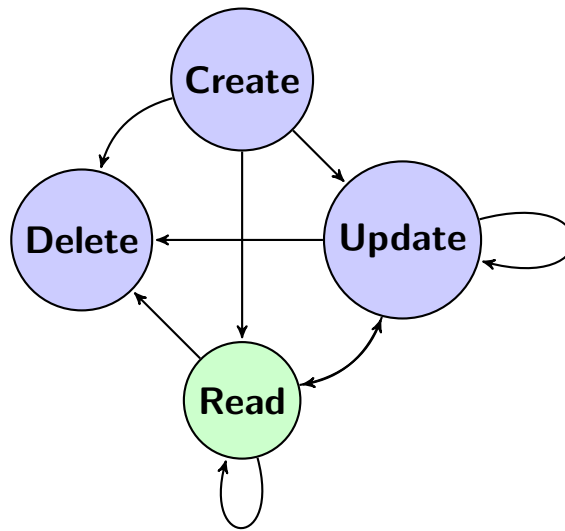


Figure 3.8: CRUD sequence

ID	attributes	event	timestamp
23	attribute1=value1; attribute2=value2	C	20150610
23	attribute1=value1; attribute2=value3	U	20150611
23	attribute1=null; attribute2=value4	U	20150612
23	attribute1=value5; attribute2=value4	U	20150613
23	attribute1=value5; attribute2=value4	D	20150614

Time  
↓

Figure 3.9: Versioning records

have been deleted before the query have an event type  $D$  and may not be considered and therefore not be included into a data set or subset.

Being able to retrieve all previous versions of a data set again entails that no data is deleted. Thus all records which cease to be valid need to be marked as deleted, but not actually deleted from the system. Obviously there exist settings in which it is either not allowed to keep all data or it is not possible to store every record for the long term. Some legal regulations also require that data has to be deleted, thus data sets which contain data to be deleted will not be available for later retrieval anymore. In cases where

actual deletion is obligatory, the affected data sets need to be annotated with appropriate metadata, indicating that they cannot be reproduced due to the deletion of records.

## R2 - Timestamping

Time is a fundamental concept which allows to structure the sequence of events based on their temporal order. Events can be put in 13 temporal relationships or intervals, as defined by [1]. Figure 3.10 shows these relations, six of which are symmetrical plus the equality relationship.

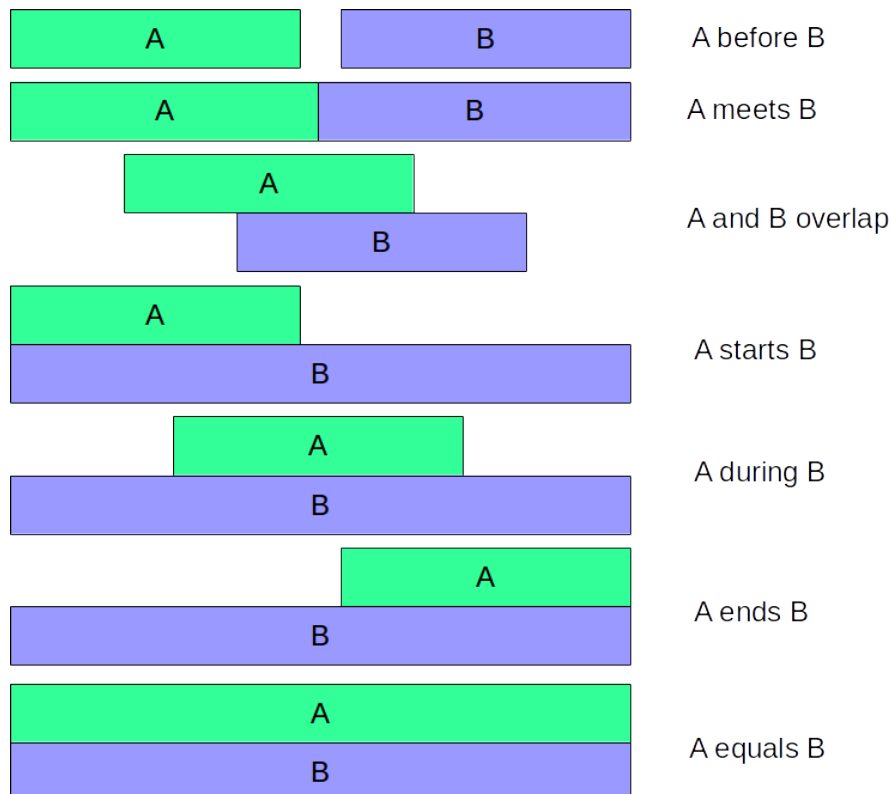


Figure 3.10: 13 time relationships by Allen

The data that we consider in this work strictly follows the temporal relationship *A before B*, which entails that all events occur in a sequential manner with no overlaps. All transactions are atomic and changes in the data do not overlap or influence each other. Therefore the data which is collected in the system can be seen as time series, i.e. the records in the data set have a natural sequence or ordering. The concept of transaction time [22] is fundamental for this approach. Upon the re-execution of a query, the system needs to consider only those records, which have been valid at the time of the query execution, i.e. the transaction. In this context, transaction time refers to the timestamp which records the operation within the database system, it does not describe the validity of the record in the real world.

In order to trace the events described in Section 3.2.2, each occurrence of an event needs to be annotated with a timestamp. A timestamp denotes the exact time and date when an event occurred. For tracing the events, the type of an event (insert, update, delete) needs to be recorded together with the time of its occurrence. Obviously there exist different granularities for timestamps, which depend on the required precision of temporal references. File systems for instance utilise different granularities for their timestamps. Whereas the FAT file system has a resolution of 2 seconds, NTFS can handle differences of 10 milliseconds<sup>14</sup>. Unix and Linux operating systems and their file systems use POSIX time, which provides a granularity of one second and counts the elapsed time since Thursday, 1 January 1970. For obvious reasons this time resolution is not precise enough for some scientific data sets, which can evolve at much higher frequencies. Many database systems therefore provide different timestamp formats and offer a large variety of granularities for their events. In our implementation we used a MySQL 5.7 RDBMS which supports timestamps of fractional seconds. Our prototype utilises timestamps of a granularity of microseconds. Details on the timestamping implementation are given in Section 4.1.3.

### R3 - Query Store

As researchers work with different data sets and highly specific individual subsets, the knowledge how subsets have been created is worth preserving. The most common way of creating a subset of a potentially large data set is to make a selection based on filtering out those records which are not needed and only include data which fulfils a given criteria. The generic approach to perform such filter operations is using a query language which allows to describe which data to be included and which records should be omitted. SQL is such a general purpose query language. For end users, interfaces exist which can hide the complexity of query languages by providing forms where users can make their selections visually or use faceted browsing. The Query Store stores these queries (thus the name) and makes them available for later reuse. Whenever a researcher uses a query (via an interface) in order to create a subset, the parameters, their order and the sortings applied can be stored. Based on temporal metadata which is collected automatically, the query can be re-executed. The Query Store has to fulfil the following functions:

- Analyse and store the queries
- Add metadata about the query including filter and selection criteria, timestamps, verification information and additional metadata such as a descriptive text.
- Attach the persistent identifier of the data source.
- Generate a persistent identifier for the query itself which is used for later retrieval.
- Compute a hash of the result set.

---

<sup>14</sup>NTFS timestamps <http://ntfs.com/exfat-time-stamp.htm>

- Calculate a hash of the query to detect duplicate queries.
- Store and provide the metadata required by the landing page.

Queries are the central piece of knowledge in this approach, as they contain all the required metadata for re-executing the query again. This includes record selection, sorting of the columns as well as the sequence of the columns. Timestamps, verification hashes or descriptions in natural language can be stored together with the query for later reference. This is very valuable information in order to reproduce experiments and verify the results. Obviously as the information is stored within the Query Store, it can be presented in a form that does not require any knowledge of the query language such as SQL. The system could also provide a tabular overview of the parameters or a transformation into any other format can be made available.

For achieving these goals, the query itself must be parse-able in a way that the parameters can be automatically extracted. Metadata such as execution times and user information needs to be integrated from the system and bundled with the query details in the Query store. This metadata on the one hand is required for re-executing and therefore retrieving the correct data set again. At the other hand it is a valuable source of additional information, such as:

- Additional descriptions how a subset was created and which parameters have been used
- Provenance data and audit log, which allows to prove when a data set was created by which user based on what version of which data set
- Long term availability of subset metadata, which can outlive the actual data as it is requires less resources to be preserved

Besides its core functionality of keeping the required metadata for query re-execution, the Query Store also preserves knowledge how a researcher has created a subset. It stores all the filter operations and their sequence of occurrence, the same is true for the sortings of a subset. The Query Store annotates all selection and filtering operations with timestamps and thus allows to investigate how a subset was selected. This is valuable additional information which allows to maintain implicit knowledge for the long term. Additionally, the metadata of the subset selection also allows to detect scientific misconduct [43], such as data fabrication or selective subsets [44]. For these reasons, the Query Store is a central piece of infrastructure serving several purposes.

The Query Store is also the point of reference for the PID system (c.f. Section 3.2.3), which establishes the persistent link between an identifier and the metadata as well as the data itself. Exactly one PID references to one specific query and its associated metadata. For easier consumption of the metadata, human users are often directed towards a landing page (c.f. Section 3.2.4). The Query Store implementation is described in detail in Section 4.1.5.

### 3.2.3 The Persistence Sub-process

After the data has been versioned, INSERT, UPDATE and DELETE events are timestamped and the Query Store has been implemented, as described in Section 3.2.2, the second phase deals with persisting subsets from dynamic data sources. Although the RDA framework allows a high degree of automation, not all queries have to be persisted per se. This entails that the decision what specific queries should be kept and stored in the Query Store remains a decision to be defined by an institutions policy. This is necessary as persistent identifiers such as DOI cannot be deleted once they have been minted. Once a query has been annotated with a persistent identifier, the identifier must resolve to usable metadata about the data set. This is not desired for all queries, especially during a testing phase where a researcher creates a new subset and cannot apriori know if the subset fulfills the requirements.

#### R4 - Query Uniqueness

In shared research environments scientists can work with the same data sources simultaneously. As the approach considers retrieving data sets from a data store via queries, it is a common scenario that researchers either retrieve one data set at multiple times or that two researchers issue the very same query. Persistent identifiers are unique by definition and by convention, each PID may only refer to one unique object. Vice versa each object should only be identified by exactly one PID within one PID system. If two researchers create exactly the same query which also returns the same result set, the system must not assign two different PIDs to identical queries. Thus a mechanism needs to be implemented, which allows to detect duplicate queries and therefore provides the basis for making an informed decision whether or not a new PID needs to be assigned.

Detecting duplicate queries is not a trivial task. Many query languages are flexible in their grammar, allowing to achieve identical results by providing different queries. SQL for instance provides different types of joins, which are either commutative or not. An *INNER JOIN* for example connects two tables based on values of the provided join predicates. This can be used to combine two tables based on identical values in the columns of each table. The *INNER* join for instance is commutative and therefore the sequence of the tables to be joined does not have effect on the result. Listing 3.2 shows an example for two distinct statements, which have an identical result.

Listing 3.2: A Commutative Join

```
1  -- Variant 1
2  SELECT
3      f.facility_id ,
4      e.experiment_id ,
5      i.instrument_id ,
6      i.measurement_value
7  FROM
8      instrument i
9      INNER JOIN
10     experiments e ON i.experiment = e.experiment_id
11     INNER JOIN
12     facility f ON f.facility_id = e.facility ;
13
14  -- Variant 2
```



```

15
16 SELECT
17     f.facility_id ,
18     e.experiment_id ,
19     i.instrument_id ,
20     i.measurement_value
21 FROM
22     facility f
23     INNER JOIN
24     experiments e ON f.facility_id = e.facility
25     INNER JOIN
26     instrument i ON i.experiment = e.experiment_id;

```

Also the sequence of conditionals does not alter the result set in SQL statements. Although the order of conditionals can have a very large impact on the performance of the processing of such queries, the results are identical.

Listing 3.3: SELECT Statements Delivering Identical Results

```

1  -- Variant 1
2  SELECT
3      f.facility_id ,
4      f.facility_name ,
5      i.instrument_id ,
6      i.measurement_value
7  FROM
8      facility f ,
9      instrument i
10 WHERE
11     f.facility_name LIKE 'DataLab'
12     AND i.measurement_value > '28.9';
13 -- Variant 2
14 SELECT
15     f.facility_id ,
16     f.facility_name ,
17     i.instrument_id ,
18     i.measurement_value
19 FROM
20     instrument i ,
21     facility f
22 WHERE
23     NOT (i.measurement_value <= '28.9')
24     AND f.facility_name = 'DataLab';

```

Detecting the equality of SQL statements is a NP-hard problem [47]. We mitigate this problem by using the interface, which provides normalised queries.

The decision if a new PID has to be assigned to a query does not only depend on the uniqueness of the query. The state of the underlying data source, i.e. if there have been changes between the execution of two queries needs to be considered. Details about this decision process are given in Section 3.2.3 and the implementation is described in Section 4.1.5.

## R5 - Stable Sorting

Research data sets are often part of a more complex scientific workflow. Thus the data produced as output of one processing step may be utilised as input in a subsequent processing step. Many experimental setups rely on the sequence of the records within one data set, thus the sorting of these records must be stable and reproducible. The sorting sequence is a property which is not always inherently fixed. Relational algebra for instance is a set based approach. For this reason many SQL concepts follow the

mathematical definition of sets. Sets in their mathematical definition do not maintain an order of their elements [24]. Thus without additional sorting specification, there is no guarantee of a stable sorting of the records of one result set.

The RDBMS MySQL supports different storage engines which are either optimised for example for reading performance (MyISAM) or referential integrity (InnoDB). MyISAM used to be the default storage engine until the version MySQL 5.5.5, later versions use InnoDB as default engine. Listing 3.4 demonstrates the effect of the storage engine with a MySQL example. First, the table is created and the storage engine of the table is set to MyISAM.

Then some test data is inserted into the table.

Listing 3.4: Creating a Simple Table and Adding Test Data

```

1 CREATE TABLE `sensorDB`.`sensors` (
2   `sensor_id` INT NOT NULL,
3   `sensor_name` VARCHAR(45) NOT NULL,
4   `measurement_value` DECIMAL(6,2) NOT NULL,
5   `measurement_timestamp` TIMESTAMP,
6   PRIMARY KEY (`sensor_id`,`measurement_timestamp` )
7 ENGINE = MyISAM;
8
9
10 INSERT INTO `sensorDB`.`sensors` VALUES (5,`temperature_lab`, 17.8, `2015-06-08
11    13:00:00`);
11 INSERT INTO `sensorDB`.`sensors` VALUES (4,`temperature_lab`, 17.8, `2015-06-09
12    14:00:00`);
12 INSERT INTO `sensorDB`.`sensors` VALUES (3,`temperature_lab`, 17.8, `2015-06-10
13    15:00:00`);
13 INSERT INTO `sensorDB`.`sensors` VALUES (2,`temperature_lab`, 17.8, `2015-06-11
14    16:00:00`);
14 INSERT INTO `sensorDB`.`sensors` VALUES (1,`temperature_lab`, 17.8, `2015-06-12
15    17:00:00`);

```

The simplified sample query in Listing 3.5 retrieves all records, MySQL returns them in the very same order as they have been inserted.

Listing 3.5: Retrieve the Data

```

1 mysql> SELECT sensor_id , sensor_name , measurement_value FROM sensors ;
2
3 | sensor_id | sensor_name | measurement_value |
4 |-----|-----|-----|
5 |          5 | temperature_lab | 11.30 |
6 |          4 | temperature_lab | 14.20 |
7 |          3 | temperature_lab | 16.90 |
8 |          2 | temperature_lab | 17.80 |
9 |          1 | temperature_lab | 19.10 |
10 |-----|-----|-----|
11 5 rows in set (0.00 sec)

```

In the MySQL versions which have been released after version 5.5.5, the default storage engine has been changed. The new default storage engine is InnoDB. In contrast to MyISAM, InnoDB applies a sorting based on the primary key of a table. This entails that after a MySQL upgrade to a newer version, the implicit sorting has been changed and the records are delivered in a different sequence. Listing 3.6 demonstrates this behaviour by changing the storage engine manually (with ALTER) and re-execute the very same query again. The records are now sorted by their primary key `sensor_id`.

Listing 3.6: The Sorting is Changed

```

1  -- Changing the storage engine
2  ALTER TABLE sensors ENGINE = InnoDB;
3  -- Re-execute
4  mysql> SELECT sensor_id , sensor_name , measurement_value FROM sensors ;
5
6  | sensor_id | sensor_name | measurement_value |
7  |-----|-----|-----|
8  |          1 | temperature_lab | 19.10 |
9  |          2 | temperature_lab | 17.80 |
10 |          3 | temperature_lab | 16.90 |
11 |          4 | temperature_lab | 14.20 |
12 |          5 | temperature_lab | 11.30 |
13 |-----|-----|-----|
14 5 rows in set (0.00 sec)

```

The effects of such updates are often not known in advance or simply not considered as the person in charge is not aware of them. For this reason the sortings of a data set must not be considered as implicit information. The same applies for more complex database technologies such as distributed storage, query optimisation or specialised index structures. All knowledge about a data set needs to be provided explicitly, otherwise the re-execution of a query to select the very same subset is not reproducible. Details how the stable sorting feature is implemented in our prototype implementation are given in Section 4.1.5.

## R6 - Result Set Verification

Being able to create specific subsets and reference them in publications or refer to them in an automated fashion during the execution of an experiment is an important benefit of data citation. Nevertheless researchers need to have evidence, that the data set they retrieved from a system, actually is the same that the creator of the data set produced. This evidence is known as authenticity [41], which is a degree how sure a user can be, that a digital record is what it claims to be. In the digital realm this is a challenging task as we cannot rely only on the bit stream of a digital object. Authenticity is a core principle in digital preservation [27]. It ensures that the integrity, correctness and completeness of a digital object can be verified. The way how authenticity is evaluated depends on the community and its interpretation of when a record is considered authentic.

In this thesis, we define a subset of data to be authentic, if it contains all the records without any changes in the same sequence as it was delivered when the user created it. For this reason the data citation framework needs to calculate a checksum or hash of the result set and store this cryptographic signature together with the metadata of the result set in the query store. The researcher can then re-calculate this checksum with the subset which was retrieved after the re-execution and then it can be verified that they are very same.

A hash function calculates a unique, fixed length string from the input data. It is a mapping between the input data to the hash value and therefore allows detecting changes to data on bit level. Figure<sup>15</sup> 3.11 depicts the mapping of a data set to the corresponding hash value.

<sup>15</sup>SHA1 image source (CC BY-SA 2.5) <https://commons.wikimedia.org/w/index.php?curid=1446602>

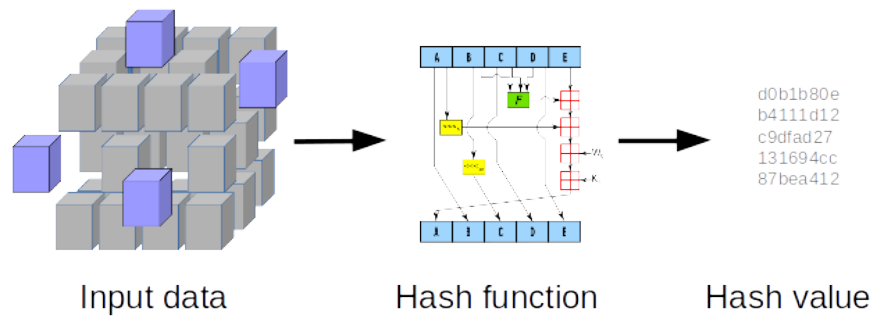


Figure 3.11: Computing a hash value

Calculating hash values of large data sets is a cumbersome task. For this reason the Query Store needs to provide several methods which allow a researcher to verify the correctness of the retrieved subset. Instead of utilising the complete result set as the input of the hash function, significant properties of the data set can be used for calculating a unique hash faster. Detecting whether or not the result set is identical needs to be simple and fast in order to increase the acceptance of the researchers. Details about the result set verification in the prototype are given in Section 4.1.6.

## R7 - Query Timestamping

In addition to data versioning and timestamping, as described in 3.2.2, also the queries need to be annotated with timing information. Whenever a user creates a sub set by executing a query against the data store, the system knows the exact execution time of the query. The timestamp of the query reflects the time at which a specific state of the database was valid. Thus it is reasonable to set the query execution timestamp to the value, which describes the state of the data rather than the actual execution time. Depending on how often the source data is updated, there are three options how to assign a timestamp. A timestamp can be assigned based on the:

- Execution time of the query
- Timestamp of the last update of the entire data set
- Timestamp of the last update of the records included in the subset

The simplest solution is assigning the timestamp of the query execution. This method is user centric, as the timing information when the user created the subset is considered to be the interesting information. The advantage of this solution is its simplicity and its performance. Storing the execution time of a query only requires to write the current

timestamp of the system into the query store. The disadvantage of this solution is that it reveals sensitive information about the user who created the data set. The timing information of a query carries private information which is then made public. This can be a limitation in some applications.

The second approach data set centric, as it utilises the last update of the entire data set as the timestamp. The advantage of this scenario is that it reflects the last state change of the data set. It can be retrieved by selecting the latest timestamp of an UPDATE, INSERT or DELETE event of the data. The disadvantage is a potentially large granularity if the data set is not updated frequently. The latest change of the entire data set must not necessarily have any effect on the subset.

The third approach considers the timestamp of the last UPDATE, INSERT or DELETE event of those records which have been included into the subset. The advantage of this approach is that it is subset centric and that it provides a timestamp which reflects the latest change of those records which have been included into the subset only. The disadvantage of this approach is that it requires to analyse the subset for its latest timestamp, which needs to be done for every query execution.

Obviously the executing time of the query can be stored in the query store as additional information, independently of the implemented timestamping strategy. Figure 3.12 depicts the three approaches.

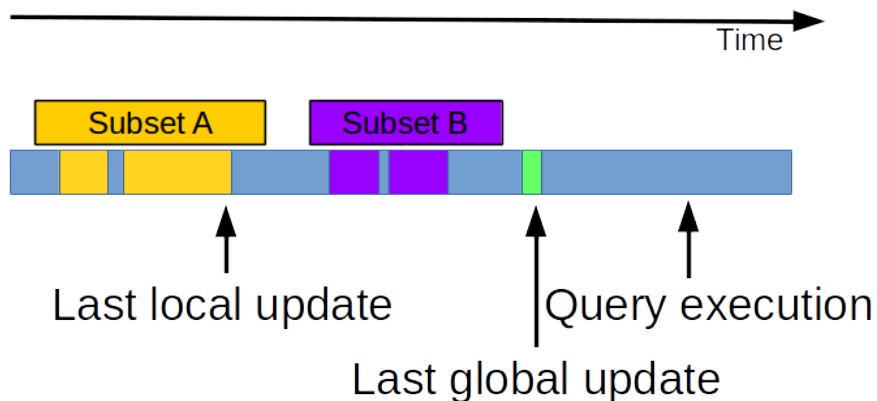


Figure 3.12: Local and global updates

The way how we can assign timestamps to the queries adds several advantages. As the re-execution timestamp can be selected freely, researchers can create a subset with the same selection and sorting criteria, but viewing a different time slice of the data. This allows comparing the effects of changes in the data with more recent or more historical versions of the same data source. As the data is versioned, the query timestamp allows to only include the data into the result set, which has been valid during the time of the query execution, thus arbitrary selections are possible, by maintaining the same characteristics of the data set.

## R8 - Query PID

A persistent identifier allows to identify a digital object unambiguously for the long term. Several systems exist, the most important implementations and approaches are described in Section 2.2. All systems have in common that they provide means of managing the link between the identifier and the object. Assigning PIDs to queries is an essential task within the data citation framework, as they allow to retrieve a query and ultimately the data set again. The PID is also used to link the data to its metadata. The resolver service of a PID system allows retrieving the metadata of the data set again and can be used for serving the landing page of a data set.

There are several different scenarios, which trigger the assignment of a new PID to a data set. The simplest case is when the query is new. As described in Section 3.2.3, the queries need to be analysed in order to detect duplicates. If no duplicate has been detected, the system needs to mint a new PID and associate the query with the new and unique string. When the query store detects a query, which has already been issued before, there are two possibilities. Either the records included in a subset identified by the query have not been changed or the data has been updated between the last execution of the same query and the current execution time. If the data has been changed, then the Query Store needs to assign a new PID in order to distinguish between the different versions of the subset. Figure 3.13 depicts the workflow of assigning PIDs

Detecting such changes in the data source is not a trivial task. Depending on the data set sizes and the data dynamics, different approaches can be applied. Details about this decision process are described in Section 3.2.3. If the data did not change between two executions of an identical query, the system needs to return the PID which has been assigned to the first execution of the query. In this case, the system does not require to calculate checksums, as the metadata already exist. Obviously in order to increase the authenticity of the data, automated checks could be issued in this case as well.

## R9 - Store the Query

After the query has been analysed, the data has been gathered and hashed (c.f. Section 3.2.3) the query metadata, user information and timing information needs to be stored persistently. In many cases the Query Store can be implemented as a database, where each query will be stored as one record. The metadata about a query can also be stored in a different system depending on the actual use case. As a minimum, the Query Store should contain the following metadata information about each query:

- Subset PID
- Super set PID (the reference to the source data)
- Data set sub-selection
- Data set sorting criteria
- Data set filtering criteria

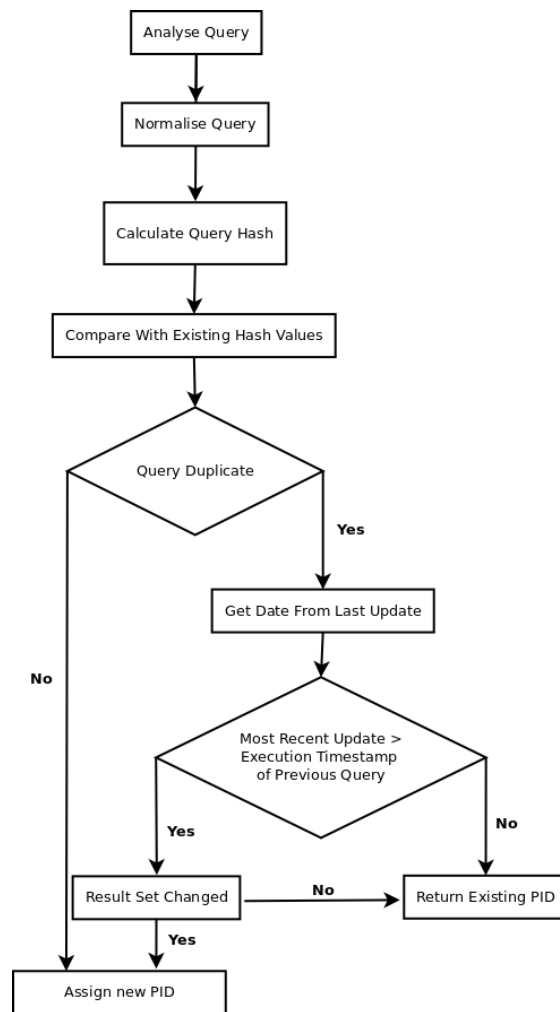


Figure 3.13: PID workflow

- Query execution time / last data (sub)set update timestamp (c.f. Section 3.2.3)
- Subset owner (e.g. the user executing the query)
- Result set hash (c.f. Section 3.2.3)

The query store holds the metadata which is displayed in the landing page of a data set. The metadata needs to outlive the source data in any case. Therefore even after the data set might be purged or not be accessible anymore, the query store needs to provide the metadata for reference. For this reason it is necessary to preserve the Query Store for the long term. In our prototype the queries are stored as a versioned record in the Query Store. Details about the query store implementation are given in Section 4.1.5.

## R10 - Citation Text

Researchers still publish most of their work in publications and therefore need to reference the data which they used in their work. Also reports often need to refer to precise data sets in order to provide evidence for the data used. Still researchers are often insecure how to cite their data or they do not know how to reference data properly [45, 13, 58]. This is a common case which hinders data reuse, although tools for supporting researchers in citing their publications and also data sets exist<sup>16</sup>. Generating citation texts automatically is a challenge, but a requirement with an increasing amount of data sets [10]. Thus in order to reduce the barriers for scientists, the Query Store needs to provide a ready to use citation text. Scientists can copy the citation text and paste it directly into their documents. The citation text references the PID of the subset and also provides the PID of the super set. Our prototype provide a citation text snippet on the landing page which can be copied and used in other documents. The citation text snippet contains information about the creator of a subset, the data set and its author as well as the date when the subset was created. Details about the citation texts are given in Section 4.1.5.

### 3.2.4 The Retrieval Sub-process

The process steps mentioned previously deal with implementing the Query Store, making the data citable, storing queries and assigning PIDs. The last process steps deal with the retrieval of subsets based on their PID.

## R11 - Landing Page

As already mentioned in Section 3.2.3, the metadata needs to be preserved for the long term. The PID assigned to the query which created the data set also links to the metadata which is stored in the Query Store in addition to each query. A main task of the data citation system is to provide a resolving service for the PIDs. The resolver service takes the PID as input and refers the user to the metadata of the subset that is associated with the PID. The metadata describes the subset in detail and allows to gain knowledge about the data set without having to download or obtain a potentially large data set in advance. The metadata of the subsets need to be human readable, which entails that the information about a subset is displayed either in a dedicated application or via a Web interface.

The most common practise for displaying the metadata in human readable form are landing pages. A landing page is the entity which a PID resolves to [4] and it provides the metadata in form of a Web page. The landing page also enables access to the actual data and provides means for downloading the data set. In many cases the landing page also provides access to previous versions of the same subset.

As an example, Figure 3.14 shows a landing page of a research data set with the DOI `doi:10.5061/dryad.234` hosted at the popular data sharing portal Dryad<sup>17</sup>. This is

---

<sup>16</sup><http://crosscite.org/citeproc/>

<sup>17</sup><http://www.datadryad.org>



the landing page displayed when the DOI was entered in the global DOI resolver service provided by the DOI foundation<sup>18</sup>. The resolver service points the browser of the user to the Dryad portal landing page for the data set<sup>19</sup>.

The screenshot shows the Dryad landing page for the dataset 'Global Wood Density Database'. The page layout includes a top navigation bar with the Dryad logo and links for 'About', 'For researchers', 'For organizations', 'Contact us', 'Log in', and 'Sign up'. Below the navigation bar, there is a 'Data from' section featuring a cover of 'ECOLOGY LETTERS' and a 'Submit data now' button. A 'Search for data' section contains a search input field and a 'Go' button. The main content area is titled 'Files in this package' and contains a table with the following information:

Title	Global Wood Density Database
Downloaded	13247 times
Description	Please direct all correspondence to G. Lopez-Gonzalez <G.Lopez-Gonzalez@leeds.ac.uk>
Download	<a href="#">GlobalWoodDensityDatabase.xls (2.047Mb)</a>
Details	<a href="#">View File Details</a>

Below the table, there is a section for citation information. It includes a box for the original publication: 'Chave J, Coomes DA, Jansen S, Lewis SL, Swenson NG, Zanne AE (2009) Towards a worldwide wood economics spectrum. Ecology Letters 12(4): 351-366. <http://dx.doi.org/10.1111/j.1461-0248.2009.01285.x>'. Additionally, it provides a citation for the Dryad data package: 'Zanne AE, Lopez-Gonzalez G, Coomes DA, Ilie J, Jansen S, Lewis SL, Miller RB, Swenson NG, Wiemann MC, Chave J (2009) Data from: Towards a worldwide wood economics spectrum. Dryad Digital Repository. <http://dx.doi.org/10.5061/dryad.234>'. At the bottom right of the citation section, there are links for 'Cite' and 'Share'.

Figure 3.14: An example landing page (Dryad)

The landing page provides human readable information about the publication which refers to the data set. It includes links to the data set download, provides information about licenses and offers bibliographic metrics. Furthermore, the portal also provides citation texts for the publication describing the results of the experiments and a dedicated citation text for the data set. Furthermore, the Dublin Core (DC) metadata terms associated with the data set are available as well. Figure 3.15 shows an excerpt of the DC terms and the contact information, Figure 3.17 shows the citation texts.

Figure 3.16 shows details about the DOI, the associated keywords and also the abstract of the publication.

Data Cite offers different services, such as statistics<sup>20</sup> about the resolved DOIs, as depicted in Section 3.18.


<sup>18</sup><https://dx.doi.org/>

<sup>19</sup><http://datadryad.org/resource/doi:10.5061/dryad.234>

<sup>20</sup><http://stats.datacite.org/>

dc.relation.ispartofseries	Ecology Letters
dc.relation.ispartofseries	12: 351–366
dc.relation.haspart	doi:10.5061/dryad.234/1
dc.relation.isreferencedby	doi:10.1111/j.1461-0248.2009.01285.x
dc.relation.isreferencedby	PMID:19243406
dc.subject	evolution
dc.subject	functional ecology
dc.subject	plant economics
dc.subject	trade-offs
dc.subject	wood
dc.title	Data from: Towards a worldwide wood economics spectrum
dc.type	Article
prism.publicationName	Ecology Letters

**Files in this package**

Content in the Dryad Digital Repository is offered "as is." By downloading files, you agree to the [Dryad Terms of Service](#). To the extent possible under law, the authors have waived all copyright and related or neighboring rights to this data. 

<b>Title</b>	<b>Global Wood Density Database</b>
<b>Downloaded</b>	13247 times
<b>Description</b>	Please direct all correspondence to G. Lopez-Gonzalez <G.Lopez-Gonzalez@leeds.ac.uk>
<b>Download</b>	<a href="#">GlobalWoodDensityDatabase.xls (2.047Mb)</a>
<b>Details</b>	<a href="#">View File Details</a>

Figure 3.15: Detailed metadata information in Dublin Core terms

Another example for a popular data sharing portal is Figshare<sup>21</sup>. Figshare also provides landing pages, which enable the preview of the data, bibliometrics and data citation texts. Figure 3.19 shows an example.

Figure 3.20 shows the details of the citation text in Figshare. Several export formats for the metadata are offered.

Although the goal of research data management is long term access, there exist scenarios in which the data may become unavailable. Legal requirements may enforce the deletion of data or financial instabilities can cause data loss. In such cases it is essential that even if the underlying source data may not exist anymore, the metadata of the subsets is preserved for the long term. This allows understanding the data at metadata level and is an important aspect for supporting authenticity of derived claims via a complete provenance record. The metadata must indicate, whether the source data

<sup>21</sup><http://www.figshare.com>

<b>DOI</b>	<a href="http://dx.doi.org/10.5061/dryad.234">http://dx.doi.org/10.5061/dryad.234</a>
<b>Pageviews</b>	6928
<b>Keywords</b>	<a href="#">evolution</a> , <a href="#">functional ecology</a> , <a href="#">plant economics</a> , <a href="#">trade-offs</a> , <a href="#">wood</a>
<b>Date Published</b>	2009-02-04T23:35:24Z

**Abstract**

Wood performs several essential functions in plants, including mechanically supporting aboveground tissue, storing water and other resources, and transporting sap. Woody tissues are likely to face physiological, structural and defensive trade-offs. How a plant optimizes among these competing functions can have major ecological implications, which have been under-appreciated by ecologists compared to the focus they have given to leaf function. To draw together our current understanding of wood function, we identify and collate data on the major wood functional traits, including the largest wood density database to date (8412 taxa), mechanical strength measures and anatomical features, as well as clade-specific features such as secondary chemistry. We then show how wood traits are related to one another, highlighting functional trade-offs, and to ecological and demographic plant features (growth form, growth rate, latitude, ecological setting). We suggest that, similar to the manifold that tree species leaf traits cluster around the 'leaf economics spectrum', a similar 'wood economics spectrum' may be defined. We then discuss the biogeography, evolution and biogeochemistry of the spectrum, and conclude by pointing out the major gaps in our current knowledge of wood functional traits.

Figure 3.16: DOI information

When using this data, please cite the original publication:

Chave J, Coomes DA, Jansen S, Lewis SL, Swenson NG, Zanne AE (2009) Towards a worldwide wood economics spectrum. *Ecology Letters* 12(4): 351-366. <http://dx.doi.org/10.1111/j.1461-0248.2009.01285.x>

Additionally, please cite the Dryad data package:

Zanne AE, Lopez-Gonzalez G, Coomes DA, Illic J, Jansen S, Lewis SL, Miller RB, Swenson NG, Wiemann MC, Chave J (2009) Data from: Towards a worldwide wood economics spectrum. Dryad Digital Repository. <http://dx.doi.org/10.5061/dryad.234>

[Cite](#) | [Share](#)

Figure 3.17: Citation texts for the publication and the data set

had to be deleted or changed and it must display if the data set is not reproducible any more due to this changes. As the metadata storage requires usually much less storage and less sophisticated storage and access solutions, preserving the metadata produces less burden for the data owner. The landing page we implemented for the prototype lists the metadata including PIDs, amount of included records in the subset, the citation text snippets and the query. The landing page provides several buttons for retrieving the different versions of a subset on demand. Details are given in Section 4.3.7.

Registrations by Allocators		Registrations by Datacentres		Registrations by Prefixes		Resolutions by Month	
April 2015							
#	Prefix	Total attempted	Successful	Failed	Total unique DOIs	Unique DOI successes	Unique DOI failures
1	10.17876 TIB.AIP	1	1	0	1	1	0
2	10.17877	4	0	4	4	0	4
3	10.17871	2	0	2	2	0	2
4	10.1184 CDL.CMU	18	7	11	7	3	4
<p>Top 10 DOIs: successes</p> <p>1. 10.17876PLATE.DR.1005 meta (2)</p>							
<p>Top 10 DOIs: failures</p> <p>1. 10.17877ECO-OUTLOOK.V0013-2-DE meta (1)                  2. 10.17877888932511846 meta (2)                  3. 10.17877888933011854 meta (2)                  4. 10.1787789782623296-4-FR meta (2)</p> <p>1. 10.17871888932511852 meta (2)                  2. 10.178718889325117171 meta (1)</p> <p>1. 10.1184744-9081-2-8 meta (5)                  2. 10.1184744-9081-2-8 meta (4)                  3. 10.1184744-9081-2-8 meta (4)                  4. 10.1184744-9081-2-8 meta (4)                  5. 10.1184744-9081-2-8 meta (4)                  6. 10.1184744-9081-2-8 meta (4)                  7. 10.1184744-9081-2-8 meta (4)                  8. 10.1184744-9081-2-8 meta (4)                  9. 10.1184744-9081-2-8 meta (4)                  10. 10.1184744-9081-2-8 meta (4)</p>							

Figure 3.18: Statistics page for human actors

## R12 - Machine Actionability

In addition to human consumption, the metadata and also the data access also needs to be machine actionable. This entails that well defined APIs exist, which allow automated agents (such as research workbenches or other scientific software) to create, locate, resolve and access data sets.

Many services such as Data Cite<sup>22</sup> already offer APIs in order to interact with the data citation system in an automated way. A common practice is to provide a RESTful API, which allows authorised and authenticated users to create and retrieve data sets and the associated metadata from the PID. The Data Cite API for instance currently supports retrieving the current URL of the landing page of a data set, listing all DOIs registered at a specified data centre, mint and associate a new DOI with a new data set or interacting with the metadata of a specified data set. Data Cite offers different other machine actionable services<sup>23</sup>, such as discovery and search facilities, content negotiation for the metadata and statistics for each data set. Other services such as Dryad<sup>24</sup> or Figshare<sup>25</sup> offer APIs for querying articles and data sets.

Our CSV prototype offers an API for retrieving the data set, re-executing queries and retrieving subsets as data download. The metadata is available via a machine actionable JSON interface. Details are given in Section 4.3.7.

### 3.2.5 Technological Change

Research data management needs to apply a long term view and accept technological change as a reality. The aim of digital preservation and its methods is to keep data accessible even when formats, software and hardware change over time.

<sup>22</sup><https://mds.datacite.org/static/apidoc?lang=en>

<sup>23</sup><https://www.datacite.org/services>

<sup>24</sup>[http://wiki.datadryad.org/DataONE\\_RESTful\\_API](http://wiki.datadryad.org/DataONE_RESTful_API)

<sup>25</sup><http://api.figshare.com/docs/intro.html>



Figure 3.19: A Figshare landing page

## R13 - Technology Migration

Technological advancements and organisational change require to migrate data between systems. This entails that the data needs be migrated into a new format, such as a new database system, a new schema or even a completely different technology. A fundamental goal of data citation is to make data sets reproducible and citable. Thus in order for data sets to sustain for the long term, the data and accompanying metadata needs to be available and accessible in the new system. This entails that there needs to be a guided process, which supports the migration itself in a reproducible way and ensures that the results of the migration are verifiably correct. This is a challenge as the access mechanism

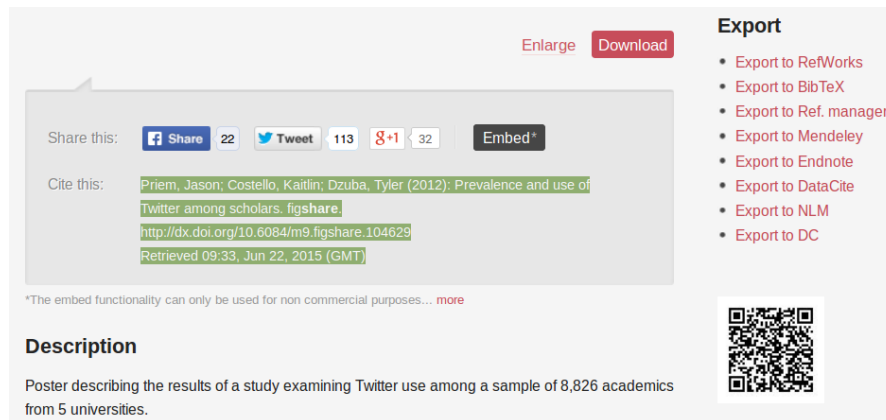


Figure 3.20: Figshare landing page details

can be very different than in the legacy system. Also the queries which are used for creating the subsets need to be migrated. Each query which was used for dynamically creating a data sets needs to be translated into the new access language and deliver the same results. This is a challenge as the compatibility of different systems and the congruence of their features can often hardly be guaranteed in advance. As mitigation strategy, the supported queries can be limited to basic structures supporting selection, projection and sorting of subsets, which have a clearly defined semantic. Thus besides a documented procedure for migrating the data and transforming the languages, open intermediate and exchange formats are preferably, which allow to be read with a broader range of tools.

In our prototype implementation we utilised well known and popular open source products. For many of the libraries and applications exist export and migration tools already, which allow to adapt to technological change. The focus is on CSV data which is already an open format and we can also export all the data stored in the database easily back into CSV.

## R14 - Migration Verification

Dynamic data citation gives the promise that the data which is uniquely referenced and persistently identified remains available for a very long term, typically the periods which are considered go beyond the lifespan of any given technology. This entails that the data needs to be migrated from the legacy system into a new technology, as described above in Section 3.2.5. As the subsets of data are created in a dynamic way based on the query on a given data source, the queries need to be migrated as well. It is crucial to provide means to verify the integrity and the completeness of all of these subsets. Preferably both systems utilise the very same hashing functions and produce identical hashes for the verification with the hash value stored in the Query Store.

In most cases this is not possible, as the underlying storage system or the representation of the records and values is completely different. The same applies for the output of

the subsets. Methods need to be applied which allow to verify that both data sets and subsets, before and after the migration process still provide the very same data sets. Also the hash key may require to be updated, as the hashing function does not work the same way between the different systems. There needs to be a comprehensible and documented mapping between the old and the new hash value and the output formats of the subsets.

### 3.3 Summary

In this section we gave an overview of the popularity of CSV data and motivated our choice for this data format in this thesis. We described how researchers use CSV files in their experimental workflows and provided an overview how CSV evolving data sets are currently cited. The approach of copying and renaming CSV files does not scale for a larger amount of data sets and hinders the reproducibility of experiments, as the process of creating subsets of evolving data is not preserved. We introduced a query based approach on versioned data, which allows to create subsets by issuing timestamped queries. With this method, we can retrieve any previously existing version of a subset without having to store individual copies thereof. This increases the flexibility and reproducibility of research data sets.

The RDA Working Group on Data Citation suggested 14 recommendations for data citation. We described these recommendations and showed how they can be used for making CSV data citable. For each recommendation, we gave concrete examples and discussed the advantages and disadvantages of alternative implementation concepts.

In the following Chapter 4 we show how we implemented the RDA recommendations for data citation in a prototype application, which allows researchers to create and cite subsets of evolving CSV data. We describe in detail, how each recommendation was considered in the implementation and provide specific implementation issues and how we solved them.





# Implementing the RDA Recommendations for Data Citation

## 4.1 An Implementation for Citable CSV Data Sets - Overview

In order to demonstrate the feasibility of the approach described in Chapter 3, we implemented a system allowing researchers to create citable subsets of structured data. The prototype supports scientists in making their data citable, creating specific subsets and assigning PIDs to the subsets, which can be resolved at any later point in time. In the following sections, we describe the practical part of this thesis.

### 4.1.1 Platform Description

The solution for creating citable CSV data sets from evolving data sources was implemented using Java EE. The Java technology stack is widely used in academia and an accepted standard programming language in many fields. For this reason we decided to use the Java ecosystem for implementing the prototype described in this thesis.

The platform we developed for enabling data citation for CSV files is organised into several components. In general the design follows a classical three tier architecture: presentation, logic and data tier. The logical tier is implemented in server side Java and provide the business logic of the application. The presentation layer interacts with the logical tier by communication via its controllers with the server component. In addition to retrieving the information requested from the data tier, the logical tier also provides all server side operations which are necessary for all steps of the data citation workflow described in Section 4.2. The logical layer is comprised by several different modules

which are used for encapsulating logically coherent tasks. It provides a CSV tool module for ingesting the CSV data and pre-processing it for the migration into the database. The database module is described in Section 4.1.3.

Each of these tiers is modularised in order to facilitate code reuse and increase the flexibility. The details of this architecture are described in the following sections.

#### 4.1.2 CSV-Tool Module

The CSV module is responsible for reading and writing CSV files. Users can upload their files into the system, which then analyses the files and reads their structure. For reading and writing CSV files, we utilise the open source library SuperCSV<sup>1</sup>. As the CSV files, their structure, amount of columns, table names etc are previously unknown to the system, existing solutions for mapping the files against Plain Old Java Objects (POJOs) was not an available solution. Thus we apply the simplest solution which allows reading CSV files line by line and parse the data as Strings into standard Java collections, such as Lists and Maps. The library is flexible enough to parse different CSV formats, independently if they are escaped, delimited with tabs or semicolons. Nevertheless the system expects that all lines have the same number of columns. A so called CSV API we developed provides public methods for parsing, cleaning, reading, writing and hashing CSV files. Furthermore the API allows adding metadata to the CSV file such as a sequence number and the insertion and last update timestamp columns. In order to migrate the CSV files into the database management system, the CSV Tools Module provides methods for reading the column headers and normalising the header names. We parse the header names and replace reserved key words and special characters with an underscore dash.

#### 4.1.3 Database Backend Module

The Database Backend Module is used for persisting and receiving the CSV data, the Query Store metadata, the persistent identifiers and the user data for interacting with the prototype. For this reason this module is at the core of the prototype implementation and it fulfils several different tasks which will be described in the following.

#### Selecting the Backend Technology

The data contained in the CSV files is stored in a relational database management system. There exists a huge variety of open source RDBMS such as MySQL<sup>2</sup>, PostgreSQL<sup>3</sup>, H2DB<sup>4</sup>, DerbyDB<sup>5</sup>, SQLite<sup>6</sup>, just to name a few. We aimed to abstract from the actually used system as far as possible by using abstraction mechanisms such as JDBC and ORM.

---

<sup>1</sup><http://super-csv.github.io/super-csv/index.html>

<sup>2</sup>[www.mysql.com](http://www.mysql.com)

<sup>3</sup><http://www.postgresql.org/>

<sup>4</sup><http://www.h2database.com/html/main.html>

<sup>5</sup><http://db.apache.org/derby>

<sup>6</sup><http://www.sqlite.org>

The generic operations required for the user and session management or the persistent identification module utilise the Object Relational Mapping (ORM) framework Hibernate ORM<sup>7</sup>. This allows changing the database management system used by these modules, by exchanging the JDBC driver in the Hibernate configuration.

The prototype we developed allows researchers to upload arbitrary CSV files and migrate the data automatically into a relational database management system. As we do not know the structure and the content of the CSV files which are uploaded by users in advance, we cannot use the Hibernate framework for storing, manipulating and retrieving the data. The reason for this limitation is that the Hibernate framework relies on mapping the database tables to Java classes. As this mapping is not available in advance, we had to write the queries used for storing, retrieving and updating the CSV data in the database backend manually with SQL.

For the sake of simplicity, we decided to use the same RDBMS or the generic tasks as well as for the data citation specific operations. We use the MySQL Community Server as the database technology, because it is widely available and easy to configure. Although there exists an embeddable version of the MySQL database software as well, we installed a dedicated database instance for performance reasons and for providing a centralised infrastructure for sharing the data sets. The MySQL server is not only used for storing the CSV files in individual tables, but it also provides access to the versioned records. The Web interface described in 4.1.7 queries the database in an Ajax-fashion, rendering the data returned from the queries in JavaScript table. As the interface provides a query-as-you-type functionality, the load on the server can be considerably high. The database module utilises a Hikari Connection Pool<sup>8</sup> for processing the queries from the Web interface and the re-execution of queries. By reusing data base connections instead of opening and closing connections for each operation, the performance could be improved significantly.

Additionally the MySQL server also holds and provides the persistent identification data and its versioned records as well. Depending on the stress of the resolver service, the database also needs to serve the landing pages and the re-execution of the queries. Thus a dedicated solution was considered more favourable.

## The Database Operations API

The database module offers high level access and methods for working with the raw data. The API of the database module provides methods for migrating the CSV files into the database tables, methods for creating subsets, re-executing queries for retrieving the data again and for interacting with the interface (cf. 4.1.7).

Users can use the Web interface for uploading CSV files. The System accepts only files with file names ending with *.csv* and stores the file after it has been uploaded in a directory on the server machine. Upon the upload of the CSV file, the system parses and analyses the file in two interactions in order to create the corresponding database

---

<sup>7</sup><http://hibernate.org/orm/http://hibernate.org/orm>

<sup>8</sup><https://github.com/brettwooldridge/HikariCP>

table for holding the CSV data. During this step, additional columns for an internal sequence number, event and timestamps are added and prefilled with the data valid at the time of the migration process. The user can define a primary key of the data set if available, or use the internal sequence key as an artificial primary key. Details about the migration process are given in Section 4.3.4, Listing 4.1 shows how the *CREATE TABLE* statement is generated from the data collected from the CSV file.

Listing 4.1: Creating a Table from CSV Data

```

1  /**
2  * Create a new table from a CSV file and drop the table if exists! Appends
3  * an id column for the sequential numbering and adds a column for the state of the
4  * record: inserted, updated, deleted
5  */
6  public void createSimpleDBFromCSV(String tableName, List<String>
       primaryKeyColumns, DatatypeStatistics datatypeStatistics) {
7
8
9       String createTableString = "CREATE TABLE " + tableName
10          + " ( ID_SYSTEM_SEQUENCE INTEGER NOT NULL";
11
12
13       Map<String, ColumnMetadata> columnMap = datatypeStatistics.getColumnMap();
14       for (Map.Entry<String, ColumnMetadata> column : columnMap.entrySet()) {
15           ColumnMetadata columnMetadata = column.getValue();
16           String mySQLDataType = createTableStatement.getMySQLColumn(columnMetadata);
17           String normalizedColumnName =
18               csvAPI.replaceReservedKeyWords(columnMetadata.getColumnName());
19
20           createTableString += ", " + normalizedColumnName + " " + mySQLDataType;
21       }
22
23       createTableString += ", INSERT_DATE TIMESTAMP(6) NOT NULL DEFAULT
24           CURRENT_TIMESTAMP(6), " +
25           "LAST_UPDATE TIMESTAMP(6) ";
26
27       // append record status column
28       createTableString += ", RECORD_STATUS enum('inserted','updated','deleted') NOT
29           NULL DEFAULT 'inserted'";
30
31       String primaryKeysString =
32           stringHelpers.getCommaSeperatedListOfPrimaryKeys(primaryKeyColumns);
33       // append primary key
34       createTableString += ", PRIMARY KEY (" + primaryKeysString + ", LAST_UPDATE)";
35       this.logger.info("Primary key is " + primaryKeysString + " and the update
36           column!");
37
38       // Finalize SQL String
39       createTableString += ");";
40
41       ...
42     }

```

The code snippet in Listing 4.1 shows how the *CREATE TABLE* statement is dynamically built by iterating over the columns. The columns and their data type are provided as input to this function after the CSV file has been parsed in the previous step. Every table has an automatically generated sequence number column *ID\_SYSTEM\_SEQUENCE* (line 9) and the two timestamp columns (line 23) for the insertion date and the last update of each record. The column name is the normalised string from the header of the CSV file (line 17). This step prevents system errors when users submit column names

with invalid characters or reserved key words.

The newly created table schema utilises an automatically iterated sequence number and it inserts the *INSERT* date during the insertion process with the current timestamp (line 23). Timestamps have a granularity of micro seconds (*TIMESTAMP(6)*, where 6 is the number of fractions). In the next step, the system iterates over the CSV file again and inserts the rows as they are encountered in the file. The operation type is stored in the column *RECORD\_STATUS*, which only accepts values of the three status types *inserted*, *updated* or *deleted* (line 27).

After the data has been migrated into the tables, each record has associated timing information. The database module encapsulates the operations with the data and therefore implements the requirements defined in Section 3.2. Each operation is recorded and annotated with a timestamp. No data is deleted and versioning is in place.

Users can insert, alter and delete data from their own data sets, by uploading a new CSV file to the existing database table. Each data set has to provide a unique primary key. With this unique key, the system is able to detect updates to existing data and can create a new record for the changed data and annotate the old version. The primary key is the only column, which cannot be changed by definition. When a user uploads a new version of the CSV file, the system checks for each record if its primary key already exists or if it is new. If the record exists, the system checks if the record has changed. If a record was updated, the system inserts a new row into the table, where the new record version has assigned the current timestamp of the event. The old record is marked as *UPDATED*, new records are inserted and marked as *INSERTED*. All operations have an assigned timestamp.

If a record is not contained in the CSV file anymore, the system detects the missing record. In case of a deletion the record is marked as *DELETED* and the timestamp of the most recent update is also set to the date of the event.

Each combination of the primary key, the event and the timestamp of the event form a compound key, which is enforced to be unique by the RDBMS. For this reason, the primary key is used to detect if a record exists in the data set already or if it was new. If a primary key is stored in the database but not contained in the CSV file anymore, the record is detected as deleted.

## **Serving the Data and Re-Executing Queries**

When a user inserted the data and has finished updating or modifying the data, the Web interface allows creating a subset, as described in Section 4.1.7. The Web interface queries the database system always for the most recent version of the data, i.e. the data which is valid at the time of the search process. In order to feed the data into the Web interface, the database module provides a RESTful Web service, which accepts requests with the query parameters. Based on these parameters, the database module writes dynamic SQL statements and executes the *SELECT* statements in order to retrieve the data from the corresponding database. Listing 4.2 shows the code for retrieving the most recent version of a data set.

Listing 4.2: Retrieve the Most Recent Version of Records

```

1      /**
2       * @param primaryKey
3       * @param tableName
4       * @return
5       */
6      private String getMostRecentVersionSQLString(String primaryKey, String tableName) {
7          String innerJoinSQLString = " FROM " + tableName + " AS outerGroup INNER JOIN
8              ( SELECT " + primaryKey + ", " +
9                  "max(LAST_UPDATE) AS mostRecent FROM " +
10                 tableName + " AS innerSELECT WHERE (innerSELECT.RECORD_STATUS =
11                  'inserted' OR innerSELECT" +
12                  ".RECORD_STATUS = 'updated') " +
13                  " GROUP BY "
14                 + primaryKey
15                 + ") innerGroup ON outerGroup." + primaryKey + " = innerGroup." +
16                 primaryKey + " AND outerGroup" +
17                 ".LAST_UPDATE = innerGroup.mostRecent ";
18         return innerJoinSQLString;
19     }

```

The response is formatted into JSON and sent to the Web interface module.

A similar approach is followed for re-executing queries. When a user wants to re-execute a query, the PID (c.f. 4.1.4) needs to be resolved. In a next step, the database module retrieves the metadata from the Query Store (see 4.1.5) and creates a dynamic SQL query which retrieves the data as it was valid during the time of the original query execution.

#### 4.1.4 Persistent Identification Service Module

The Persistent Identification (PID) Module is responsible for establishing a permanent link between an identifier and the metadata of a query or subset respectively. Details about PID systems in general are described in the related work Section 2.2. As at the time of writing of the thesis no PID prototype or mock-up service existed, we implemented our own PID service prototype. The PID Module consists of several sub-modules which group the functionality.

##### The PID API

The PID API offers publicly accessible methods for creating, retrieving and managing the persistent identifiers. The PID module utilises a separate user management, allowing to associate users with their organisational prefix and store additional metadata about the user who created the PID for the long term. Currently the system offers three different types of identifiers: alphanumeric, alphabetic and numeric. The following list shows examples of such identifiers for each type for the organisation having the prefix 6789.

- Alphanumeric: 6789/89AJBE0CL5mX
- Numeric: 6789/31396428315789
- Alphabetical: 6789/aqESfsplLrTzxdA

Each organisation may define constraints in terms of capitalisation of the identifier strings and in terms of length. The uniqueness of the identifiers is enforced by the RDBMS integrity constraints, as the primary key of the persistent identifiers table is a compound key of the *prefix* and the *suffix*. Thus it is ensured that all identifiers stored within one instance of the PID service module are unique. Global uniqueness can be ensured by defining the URL of the resolver service, which serves as a further prefix, where the uniqueness is enforced by the Domain Name System (DNS).

The PID Service provides methods for creating alphanumeric, numeric and alphabetical identifiers on demand. The three types inherit their identifier generator methods from the *PersistentIdentifier* super class and implement the specific method required. The methods for generating alphabetical and alphanumeric strings utilise the Apache Commons String Utils library<sup>9</sup>, the numeric type utilises the standard Java random number generator. PIDs can be hierarchical in order to express inclusion relationships, such as super set and subset. One super set can be the parent of several subsets. For this reason we introduced a structure allowing to define such hierarchical relationships of PIDs, which allow defining and resolving parent/child relationships.

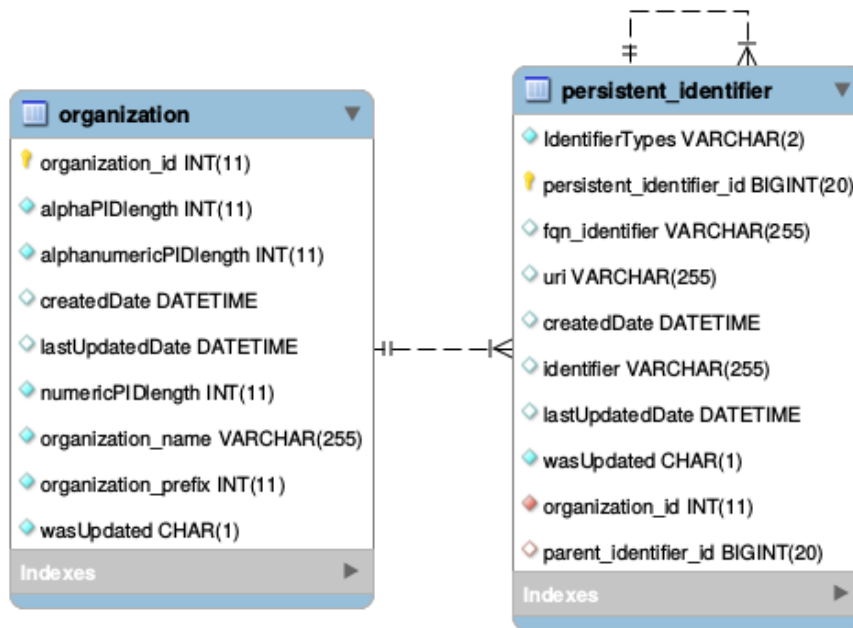


Figure 4.1: ER diagram of the PID system

Figure 4.1 depicts the entity relation (ER) diagram of the PID table. It shows the self-referential relationships of the PID table, allowing to create hierarchical PIDs of arbitrary depth. In order to avoid table joins during the resolving process, we implemented the three types in one table and utilise one of the discriminator values *A, AN, N* for indicating

<sup>9</sup><http://commons.apache.org>

the alphabetical, alphanumeric or numeric nature of the identifier string. Each identifier has a prefix, which is a foreign key to the organisation table. This key establishes the relationship between the two entities and allows defining not only metadata about the organisation issuing the PID, but also defining organisational wide constraints such as the identifier length. The PID table stores the identifier and the fully qualified name (*fqn\_identifier*). The full qualified name consists of the prefix and the identifier. It is used in order to speedup the resolver process, as the resolver service directly queries the identifier including the prefix and also the URI.

The PID system is fully audited, using the audit framework Hibernate ORM Envers<sup>10</sup>. This is essential, because the persistence of the system is established by allowing operators to update the resolver URI and therefore to change the location of the data. The API only provides methods for updating the URI, for obvious reasons, the identifier string cannot be changed. All changes are traced within the audit tables, as depicted in Figure 4.2.

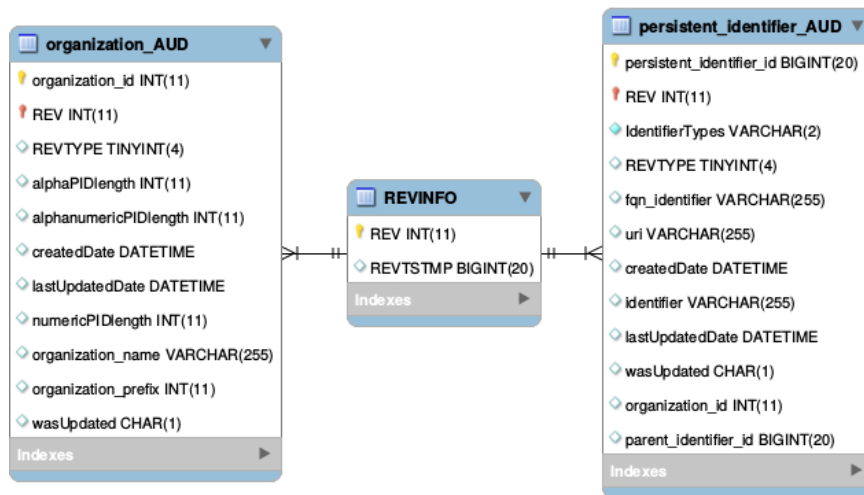


Figure 4.2: Audit tables of the PID prototype

Each operation of the PID module is thus timestamped and automatically versioned. This allows tracing the changes of the PID metadata, especially of the location (URL) of the data, and thus supports the long term view of the system.

## The Resolver Service

The PID API also offers methods for resolving the PIDs and referencing the metadata of the data sets. The system stores whether a PID refers to a super set or a subset. Once a user uploads a CSV file into the system, the PID Service attaches a new PID to the super set, by referencing the corresponding table. The approach is also query based, as

<sup>10</sup><http://hibernate.org/orm/envers>



the Query Store (c.f. Section 4.1.5) stores the standard *SELECT* query, which includes all columns in the order as they have been in the file, sorted by the artificial primary key which correlates with the record order in the file. Upon the resolution of a super-set, the system retrieves the full table at the point of execution again. For resolving a subset, the system checks for the availability of the super set and re-executes the query based on the specific parameters of the subset. As all the data and the queries are versioned, the corresponding data can be retrieved.

The resolver service is implemented as a RESTful Web service, which accepts requests from authenticated users and responds with JSON. The Resolver Service provides its own API and allows querying metadata associated with the PID from the system. The provided URI (which is a URL in the practical implementation) points to the actual data repository which can either provide the landing page (cf. Section 4.1.7) or with a data download. The API of the resolver service provides additional methods for retrieving all PIDs associate with an organisational prefix, resolving parent and child relationships in both directions and for retrieving the metadata of the PID.

#### 4.1.5 Query Store Module

The Query Store Module is the core of this project, as it stores all the metadata of the subsets by persisting the query. The Query Store is located in a separate database schema and provides an API exposing public methods interacting with the system.

##### The Structure of a Query

In order to store the query and its metadata, we created a *plain old java object* (POJO), which contains the appropriate fields, and wired it with Hibernate ORM to a relational database schema. The ER diagram of the schema is depicted in Figure 4.3. The central entity of the Query Store is the table *query*. It stores an internal *queryId* and the associated persistent identifier *PID*. This identifier is minted from the PID service and is unique within one database instance, which is enforced by integrity constraints in the database and by the Hibernate ORM model. Each query has an associated creation timestamp, which is stored in the field *createDate* as well as an *execution\_timestamp*, which is set to the exact time the original query was executed. The *data\_source* field references the parent data set, which is the database table created from the CSV file. The PID therefore uniquely associates a query to a specific data set. Further information about the parent data set is stored in the entity base table, which is described in Section 4.1.5. Further query metadata is a textual description *query\_description*, a hash of the query *queryHash* and the query string *queryString*. The description is manually entered by the user via the Web interface. The description can be displayed on the landing page and provides information mainly for human consumption.

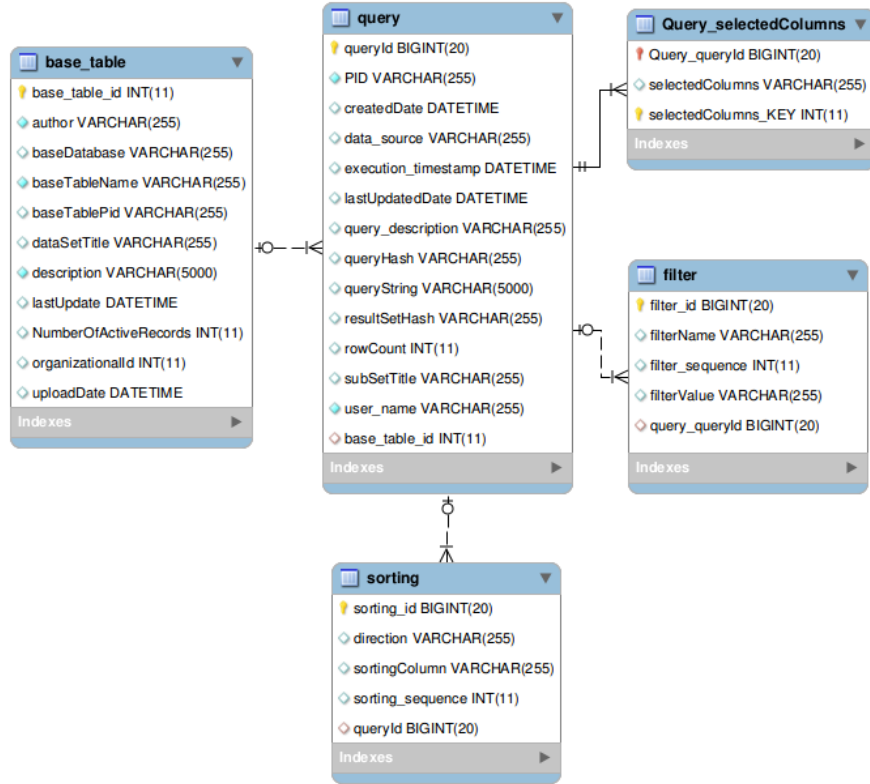


Figure 4.3: Query Store ER diagram

## Detecting Duplicate Queries

The query hash is essential for detecting duplicate queries. As described in Section 3.2.3, a PID uniquely identifies one unique object. Queries are considered new, if they have not yet been issued by a user before. In this case the query gets a new PID assigned. If a query has been issued before, the existing PID needs to be returned, if the result sets have not changed since the previous invocation. If the data changed between two executions of the same query, the system also needs to assign a new PID in order to reflect that these two results sets are in fact different objects. A query is only considered identical, if the very same query parameters have been used. The sequence of columns in a data set for instance is a crucial property, as otherwise the result set is considered different. The same applies for the sorting, where also the sequence of sortings is considered an important property. The third important property which renders a query result unique is its filters. In contrast to the columns and the sortings, the sequence of the applied filters, however is not considered important, as the application only allows conjunctive lists of properties instead of precedence expressions. The prototype interprets all filters as connected with an *AND* conjunction. Thus the order of applied filters does not have any effect on the result. The way how the system reports on the assignment of PIDs

does have privacy issues.

For detecting query duplicates, the system could also use the SQL string which was sent to the database system and compute a hash thereof. Although this approach allows detecting syntactically different SQL query strings, it does not detect semantically equivalent queries as such. Examples for such queries have been given in the Listing 3.3 in 3.2.3. Instead of relying on the query string, the system concatenates the PID, the selected columns, the applied sortings and the alphabetically sorted list of filters into one string. This string is then used as input for the SHA1 one-way hash function which produces the unique hash key. As the sequence of filters is automatically sorted, even filters entered in a different sequence lead to identical results and therefore hash keys. Listing 4.3 shows the method for computing the hash key based on sortings and filterings. As shown in line 26, all filters are retrieved from the query store and appended to one string. We use the insertion order of the filters to maintain a reproducible sequence of filters for the hash key computation. The same technique is used for the sortings, as shown in line 42, where all sortings are concatenated.

Listing 4.3: Computing the Query Hash

```
1      /**
2      * Iterate over filters and sortings and calculate a unique hash.
3      * Sortings and Filterings are stored in LinkedHashSets.
4      * Their insertion order is preserved. If a query uses a different
5      * order of insertion, then the hash will be different.
6      * If this behaviour should be changed, use a TreeSet and sort it
7      * alphabetically.
8      *
9      * @param query
10     * @return
11     */
12     private String calculateQueryHash(Query query) {
13         // concatenate all query details: data source pid + filters + sortings
14         String queryDetails = "";
15         String allFilters = "";
16         String allSortings = "";
17
18         // Append data source PID
19         queryDetails = query.getDatasourcePID();
20
21         // Iterate over all filters, concatenate their keys and values and normalize
22         // the string.
23         List<Filter> filters = query.getFilters();
24         if (filters != null) {
25             if (filters.size() > 0) {
26                 Iterator<Filter> filterIterator = filters.iterator();
27                 while (filterIterator.hasNext()) {
28                     Filter filter = (Filter) (filterIterator.next());
29                     allFilters += filter.getFilterName() + filter.getFilterValue();
30                 }
31                 allFilters = this.normalizeString(allFilters);
32             }
33             // append all filters
34             queryDetails += allFilters;
35         }
36
37         List<Sorting> sortings = query.getSortings();
38         if (sortings != null) {
39             if (sortings.size() > 0) {
40                 Iterator<Sorting> sortingIterator = sortings.iterator();
41                 while (sortingIterator.hasNext()) {
42                     Sorting sorting = (Sorting) (sortingIterator.next());
43                     allSortings += sorting.getSortingColumn() + sorting.getDirection();
44                 }
45             }
46         }
47     }
```

```

45         }
46         allSortings = this.normalizeString(allSortings);
47     }
48     // append all sortings
49     queryDetails += allSortings;
50 }
51 // Calculate the hash of the text
52 String uniqueQueryHash = this.calculateSHA1(queryDetails);
53 return uniqueQueryHash;
54 }

```

## Verification Metadata

In addition to the previously mentioned metadata, the system also stores the plain text SQL string, which was executed. This allows storing how the system originally retrieved the data and users with direct SQL access to the database could retrieve the same data set manually. Additionally for detecting the uniqueness of queries for the assignment of the PIDs to the query, the result set itself needs to be verifiable. For this reason each query stores a hash of the result set. Details how the hash of a result set is computed are given in Section 4.1.6.

## Storing Parameters: Columns, Sorting and Filters

Each subset consists of a set of columns, as specific sorting of the said columns and the filters selected by the user. These parameters have in common, that several instances of each of these entities can occur in one query. Therefore the Query Store utilises a one to many mapping between the query and the sorting, filter and column entities. The columns are stored per query in the sequence in which the user selected them in the Web interface. The same applies for the sorting, which stores the sorting directions *ascending* or *descending* for each column. Filters additionally store the filter value, which was applied to a specific column. Filters are connected by the Query Store in a conjunctive form (with *AND*). Their sequence is stored for reproducibility reasons and for understanding in what sequence a researcher entered the filters.

The Listing 4.4 shows the method for adding a filter. First, a new *Filter* object is created and the filter name (key) and the criteria (value) are set (lines 10). Then, the current list of filters for the specified query are retrieved and the newly created filter is appended at the end of the list (line 15). The query object is updated (line 16) and the query hash is recalculated including the newly added filter (line 20). Both operations are executed within one transaction using Hibernate ORM.

Listing 4.4: Adding a Filter

```

1     /**
2     * Add a new filter to the query. Recomputes the query hash.
3     *
4     * @param query
5     * @param key
6     * @param value
7     */
8     public void addFilter(Query query, String key, String value) {
9         // create new filter and persist it
10        Filter filter = new Filter(query, key, value);
11    }

```

```

12     this.session = HibernateUtilQueryStore.getSessionFactory().openSession();
13     this.session.beginTransaction();
14     this.session.saveOrUpdate(filter);
15     query.getFilters().add(filter);
16     this.session.saveOrUpdate(query);
17     this.session.getTransaction().commit();
18
19     // recalculate query hash
20     String newQueryHash = this.calculateQueryHash(query);
21     query.setQueryHash(newQueryHash);
22
23     this.session.beginTransaction();
24     this.session.saveOrUpdate(query);
25     this.session.getTransaction().commit();
26     this.session.close();
27 }

```

## Further Metadata

In addition to the parameters and main metadata bits, the Query Store also contains statistics about the result sets, such as a row count. This allows users to quickly estimate the amount of data which will be returned by a request. The system also stores the user name which is unique within one instance and allows to save which authenticated user created a data set. This does not necessarily need to correlate with the user which is authorised with the PID system. A short title can be used for describing the data set more briefly than with the textual descriptions. Furthermore the short text can be used for automated citation text generation, which will be described in Section 4.1.5. The design of the Query Store permits storing any arbitrary metadata additionally to the mentioned fields. Thus the proposed solution is extensible and can be adapted to the requirements of its users.

## Security Considerations for PID Assignments

The workflow presented in Figure 3.13 in Section 3.2.3 describes the decisions to be made, whether or not a new PID has to be assigned to a query. Users could gain knowledge about the work of peers, when they discover that the very same data set has been issued earlier. This can bring out a privacy issue as it allows to derive what kind of data set a person was interested in. In areas where researchers work competitively, the PID system can be exploited for learning which precise data set another person was using. Thus there exist scenarios where the system may not disclose that a query has been issued before to the user. For preserving the privacy of the subset creation process, the system can always assign a new PID, but link the new PID to the original query, allowing administrators understanding the relationships between data sets. Alternatively, the system can also prevent resolving sensitive data sets before they have been made public or it can also hide metadata, which would reveal details about the user who created it.

## Base Tables

The system needs to differentiate between the super set, which corresponds to the CSV file by referencing the complete table. Figure 4.3 depicts the *basetable* entity, which is

responsible for referencing the full data set from a subset. The basetable entity stores the name of the database scheme a data set is located in and the table name which stores the raw data and revisions of the data set. The combination of database scheme and table name allows identifying a data set uniquely within one database instance. Additionally, each basetable record has its own PID assigned, which allows resolving the location of the table for the long term. Each data set is described in the basetable as a record, which contains a data set description, the creation date (i.e. when the CSV file was uploaded), a title, the date of the last update of any of the records in the data set and the amount of active (i.e. not deleted) records.

### Citation Text Creation

Both the parent data set (c.f. 4.1.5) and each subset (c.f. 4.1.5) have associated metadata stored with each record or query, respectively. A user can provide a textual description, PIDs and timestamps are assigned automatically. This metadata can then be used for generating a citation text, such that users can easily copy the text into their reports, papers, documents or Web sites. These texts are very important for the acceptance of data citation methods, as users tend not to cite data if they are perceived as a burden for them to use. Therefore it is crucial to lower this barrier as much as possible. Figure 3.17 shows an example from a citation text. For this reason, the prototype we developed provides automated citation text creation for each record on its landing page. Figure 4.4 shows the citation text of the super set as it is displayed at the landing page. In Figure 4.5 the citation text of the subset is shown. It references the super set by including its PID and the timestamp of the subset creation.

Suggested citation text: Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere (2015): " The Million Song Dataset (MSD)", PID [ark:12345/YEcHQ2legb]

Figure 4.4: Citation text of the super set

Suggested citation text: Stefan Proell (2015) "Patterns in Classical Music" created at 2015-07-15 11:47:47.0, PID [ark:12345/MtUAdMZ411]. Subset of Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere: " The Million Song Dataset (MSD)", PID [ark:12345/YEcHQ2legb]

Figure 4.5: Citation text of the subset referencing the super set

Researchers can easily copy and paste these citation texts in their publications, just by browsing the landing page of the data set.

#### 4.1.6 Result Set Verification Module

For increasing the trust in research data and providing proof for the correctness of the data sets, the system needs to give researchers the means to verify that the data which they obtained is actually correct. The ResultSetVerification Module provides an API with methods for verifying the correctness and completeness of the result set. When we speak about correctness in this context, we mean that no data has been changed or added between the first execution of a query and of its subsequent re-executions of a

subset. Completeness entails that no data has been removed. Thus the data needs to be exactly the same as it has been at the first execution, including the sequence of columns and sortings.

## Selecting the Hash Functions

Again hashing functions are the tool of choice as they allow to calculate a unique string from the input which can be compared for equivalence with the provided hash key from the landing page. Several approaches exist for hashing result sets. One possibility is to export the result set into the CSV format and calculate a hash sum server side in the file system of the server machine by using standard tools such as md5sum, shasum or other hash functions provided by the operating system.

This approach requires to export each data set and subset first into the CSV file format. Obviously this requires considerable resources in terms of storage space and computation time. For large data sets the calculation of a hash sum can consume a considerably long time, thus rendering the approach unfeasible for many situations, where users expect fast answers to verification checks. Also the hash is then depending on the CSV file format, which may not always be the desired target format. Thus we decided to move the hashing of the result set into the application and provide verification methods without the need for prior data export.

Computing the hash in the database is preferable in terms of computation time, as no data would need to be transferred between the server and the client. Unfortunately most RDBMS would require a custom stored procedure for calculating a hash sum of a complete result set, as there do not exist standardised functions for computing checksums of result sets server side. As we would like to avoid database vendor specific functions and code as much possible, we decided to move the hashing into the Java application, where necessary changes can be mapped into different SQL dialects in a central place. In order to enable flexibility when it comes to the selection of the hash function, can be adjusted as depicted in Listing 4.5.

Listing 4.5: Initialize the Crypto Module

```
1 private MessageDigest initCryptoModule(String algorithm) {
2     HashSet<String> algorithms = new HashSet<String>();
3     algorithms.add("SHA-1");
4     algorithms.add("MD5");
5     algorithms.add("SHA-256");
6     if (algorithms.contains(algorithm)) {
7         this.crypto = null;
8         try {
9             this.crypto = MessageDigest.getInstance(algorithm);
10            this.crypto.reset();
11        } catch (NoSuchAlgorithmException e) {
12            e.printStackTrace();
13        }
14    }
15    return this.crypto;
16 }
```

## Computing a Verifiable Hash from a Potentially Large Dataset

Research data sets can become large in terms of storage demand and therefore computing a hash for a several GB large CSV file becomes unfeasible. The same is true for large database tables and for result sets as well. For detecting changes on record level, the system would need to compute a row based hash including all fields of each record. Whenever a record is updated or deleted, the hash key must be recomputed. In order to verify the integrity of each record, the result set hash must be computed based on the concatenated row hashes for each subset. As this operation requires considerable more resources, it was omitted for this implementation.

In order to implement a practical solution we decided to compute hash keys based on the features which allow us to measure the uniqueness of a data set without having to include all records. Our approach is based on the assumption that the RDBMS provides atomicity, consistency, isolation and durability of the data it stores in accordance to the ACID principles [25]. Our implementation is built on top of MySQL<sup>11</sup>, which provides with the InnoDB Storage Engine a system with high ACID compliance.

Manipulating the data is only allowed by using the API of our solution. Thus changing single values without leaving traces is not enabled within the application. For this reason our approach utilises a simplified hashing scheme, allowing to store minimal information for verifying the correctness of the data set. Figure 4.6 shows an schematic overview of the hashing scheme, based on an example of the million song data set (MSD) [7]. The figure depicts a subset of Mozart songs, where the columns *title*, *artist\_name* and *duration* have been selected. The songs in the subset are sorted by duration.

Dataset PID					
title	song_id	release	artist_name	duration	timestamp
Monday Monday	SOLZHTF12A8C136AB9	California	Wilson Phillips	196.8322	20160201
Ecco Il Birbo Che Tha Offesa	SOABXSF12A8C1389B2	Don Giovanni	Wolfgang Amadeus Mozart	313.52118	20160530
Utopia [Feat Chris Jones]	SOYVRJN12A8C144F71	Utopia	Within Temptation	236.52223	20160604
Lügner	SOMALFU12AB017E1A4	Steh grad	Wolfgang Ambros	211.40853	20140708
Bronze Silber und Gold	SOCWXOZ12A8C137973	30 Jahre	Wolfgang Petry	193.51465	20151103
It's Too Late	SOLVBQQ12AF72A2299	It's To late	Wilson Pickett	192.86159	20160804
Serenade In Strings	SOIMTS12AB0188D6B	Classical Selections	Wolfgang Amadeus Mozart	113.68444	20160723

Figure 4.6: Example subset with verification information

As the PID is unique by definition, it identifies each data set unambiguously. The list of columns and their sequence define the structure of the CSV data set. The list of selected columns is appended to the PID of the data set as a string. For verifying which records have been included into the data set, we store a list of all the records of the result set in the order in which they have been retrieved. As each record requires having a primary key and primary keys are unique by definition, it is sufficient only considering the primary key of each record (in the example in Figure 4.6, the column *song\_id* is the primary key). As we store the sequence of these records, we can verify the order of the

<sup>11</sup><https://dev.mysql.com/doc/refman/5.7/en/mysql-acid.html>



records in the data set, thus re-sorting or moving records leads to a different hash. For ensuring that the relevant version of each record is included, we append the timestamp to the primary key, which ensures both, uniqueness and the appropriate version to be considered.

Figure 4.7 depicts the input for the hash function used to calculate unique, individual hash keys for each data set and subset. For calculating the hash of a subset, the system retrieves the data set by re-executing the query stored in the Query Store. The application retrieves the PID of the subset and the list of selected columns from the Query Store. Both metadata are stored in a string by concatenation. In a next step, the system iterates over the result set. For each record contained in the result set, the system appends the primary key column and the timestamp of the record in a string and adds the string at the end of the PID and column list string.



Figure 4.7: Input of the hash function

The string prepared by the verification module serves as input for the hash function. As hash functions create hash keys of fixed length, the hashing scheme does not require permanent storage space, but rather main memory for its computation. Equation 4.1 depicts the input size for the hash function, where  $len$  is the function denoting the length of the string in characters. The hash input string consists of the unique  $PID$  string, the ordered list  $CN$  of column names as concatenated string, the primary key  $PK$  and  $TS$  the timestamp in a string representation, for instance in Unix time [33].

$$len(input) = len(PID) + \sum_0^i len(CN_i) + \sum_0^i (len(PK_i) + len(TS_i)) \quad (4.1)$$

The PID does not have a technical limit regarding the length of the identifier string, which is composed of the prefix and the suffix. Typically, the prefix contains six characters (including the “.”) and a suffix which is organisation specific. Considering that a PID could also map a generic identifier such as ISBN, the average length of the suffix is assumed to be 13 characters, summing up to 19 characters for the PID. As the data and the metadata is stored in a database management system, the limitations of these also apply to the CSV data we want to store. Column names have a technical limit which is imposed by the RDBMS, in our case MySQL. MySQL does not allow column names which exceed the length of 64 characters. Technically, MySQL can only store 1017 columns per InnoDB table and store a virtually unlimited amount of records. As our system provides automatically incremented sequence numbers for the records, the hard limit of records is set by the size of the unsigned BIGINT<sup>12</sup> column type used for

<sup>12</sup><http://dev.mysql.com/doc/refman/5.7/en/integer-types.html>

the sequence number column<sup>13</sup>. When storing all values in VARCHAR columns, the theoretical maximum length of each record is 65 535 characters<sup>14</sup>. Changing the default column data type to a textual type allows increasing this limit arbitrarily.

In comparison, the current version of Excel allows 1 048 576 rows by 16 384 columns<sup>15</sup>. Column headers may have a length of 255 characters and each cell can contain up to 32 767 characters. In order to avoid errors, the system automatically caps column names above this length and introduces artificial column names if necessary, which is documented with messages in the log file.

For describing the best, average and worst case for the computation of the hash key, it is assumed the processed CSV files either are limited by the capabilities of Excel or of MySQL, whatever limit is reached first. The average case is described with the million song data set [7], which serves as the basis for the evaluation in Section 5.1. The data set contains as the name suggests one million records, which are described in 23 columns. Table 4.1 provides the storage amounts in bytes as used by the internal storage requirements for MySQL.

	Minimum Case	Average Case <sup>16</sup>	Worst Case
PID	8 B	17 B	32767 B
Column names	2 B	190 B	5085 B
Primary keys	4 B	19 000 000 B <sup>17</sup>	3 221 225 472 B <sup>18</sup>
Timestamps	1 B	4 000 000 B	8388608 B
Sum	15 B	23000207 B ( $\approx$ 23MB)	3229651932 B ( $\approx$ 3.23GB)

Table 4.1: Storage requirements in bytes

Based on the technical limitations for the database system and the assumption of Excel as a data export source, the input string for the computation of the hash function cannot exceed a length of 3229651932 bytes, which roughly correlates to the size of a DVD ISO image. Hashing images of such size is a common task, which can be performed on commodity hardware within reasonable time. Several different hash functions exist, as described in 4.1.6. The hash functions which are available via the API in this implementation and their output strings are provided in Table 4.1. The computational time has been measured by executing the command line tools on a 3.2 GB large test file within a VirtualBox Machine with 2 GB RAM. For comparison it is worth stating that the MSD data set of song metadata used as an example in this thesis utilises 205 MB of storage as CSV file.

The minimum case assumes that exactly one record needs to be cited. Thus the input for the hash function is minimal and only contains a short PID, a one letter column

<sup>13</sup>Hard limit for BIGINT: 18 446 744 073 709 551 615 records

<sup>14</sup><http://dev.mysql.com/doc/refman/5.7/en/char.html>

<sup>15</sup><https://support.office.com/en-za/article/Excel-specifications-and-limits-ca36e2dc-1f09-4620-b726-67c00b05040f>

<sup>16</sup>Based on the MSD test data set

<sup>17</sup>The primary key column track\_id in the data set is 18 digits long and 1 one byte overhead

<sup>18</sup>1048576 records, each having a maximum PK length of 3072 bytes

Function	Length (Hex)	Time (sec)	Example
MD5	32	9.2	5867d2a43a7c3e2acf987d6c39269bf7
SHA1	40	14.7	05d717c57e8f9cf048928a6e66ee3f935fb97f92
SHA-256	64	23.7	188045453e8...f5bb5ee107f06d11025a52
SHA-512	128	16.7	9b18513a96...5abe9191b876ef29b4...1da

Table 4.2: Hash outputs

name (plus one byte overhead) one Integer of 4 bytes for the primary key and a coarse granular timestamp (one year) stored with one byte. The average case assumes that the user selects all columns from the file and includes all one million primary keys, where each key occupies 18 bytes plus one byte overhead. The timestamp has a granularity of one second (Unix time) and requires 4 bytes each. The worst case scenario assumes a (unrealistically) long PID, the maximum columns and names length per column which MySQL can handle as well as the most primary keys Excel process. Timestamps support fractions of seconds and utilise 8 bytes each.

The hashing scheme in this scenario allows to verify the completeness including the sorting and sequence of columns for the data. Even for considerably larger data sets, the hashing scheme produces string based inputs which can be handled by commodity hardware. By only considering the primary key, the timestamp and the sequence of each result set, we can produce a hash key which allows detecting changes in the sequence or amount of data returned. The system does not yet implement means for verifying that the content of the records did not change in a manipulative way. As users may only interact with the system by using the API and their defined interfaces, manipulation must occur on database and system level and is therefore beyond the scope of this work.

A special case is the empty set. Two queries on the same data set selecting the same columns but utilising different filters, which both deliver an empty result set, would produce the same verification hash. The purpose of this mechanism is to detect changes between subsequent executions of the same query, but not to identify subsets based on this hash. For this reason the fact that two non-identical queries can produce the same verification hash, if both deliver the empty result, does not contradict the principles of this approach. The hash still allows to detect changes within each subset individually.

The computation times presented in Table 4.2 only consider the hashing itself. For utilising the hashing in our implementation, the system needs to retrieve the data from the database system and concatenate the records of the result set one by one, before the data can be processed. The time required for retrieving the data depends on the location of the RDBMS (local or remote) and the transfer times of the data.

#### 4.1.7 Web Interface Module

As described in Section 1.2 researchers require incentives in order to share and cite their data. Providing a usable interface is one essential component which is fundamental for user acceptance of a solution. There exists a wide area of user interfaces, ranging from

low level command line applications via pragmatic and minimalist interfaces towards sophisticated and elaborated applications specifically designed and adapted for a defined set of devices.

As the aim of this work is to lower the barrier for researchers to create citable subsets, several considerations were taken into account. Using the citation framework should be supported by an application which can be used by a broad range of users. As installing dedicated software products is often limited to institutional policies, we decided to utilise a Web application for providing the interface. Hence, no additional software is needed and the application can be deployed centrally on a dedicated server.

The interface is realised as a JSF application which is deployed in a Tomcat servlet container. The servlet container executes the server side Java code and renders the view for the user interfaces. The Web application provides separate interfaces for the interaction with users and it is realised by the use of facelets. Several different open source implementations of the JSF standard exist, we utilise the reference implementation Mojarra<sup>19</sup>.

In order to have a clear separation of the code, the implementation follows a model view controller (MVC) pattern. The view renders the data provided from the model, the exchange of data and states between view and model is managed by the controller components. In order to provide a standardised user interface, we utilised the JSF framework PrimeFaces.

The Web interfaces is used for handling the workflow for CSV data citation and supports the user during the necessary steps. In order to facilitate multi-user support, we implemented a session management for the Web application. The system requires users to authenticate against the system, by providing a user name and a password. After successful login, the application monitors the state of the user session and ensures that users do not interfere with each other.

## 4.2 Supported Workflow

The workflow which structures the steps are described in the following sections. We utilise a user centric view in these sections in order to describe the visible features and end user experience.

1. Login
2. Main menu
3. Provide metadata
4. Upload data
5. Migrate CSV into database table
6. Create subset

---

<sup>19</sup><https://javaserverfaces.java.net>

7. Publish data set
8. Verify data set
9. Resolve PIDs
10. Obtain data set

## 4.3 Platform Details

In the following sections we provide details of the implemented solution.

### 4.3.1 Login

The system is designed for supporting multiple users. For differentiating between the different individuals, each user needs to authenticate against the system. Each individual works for one specific organisation, therefore each user has one assigned institutional prefix which forms the first part of the persistent identifier as described in 2.2.

The metadata of the user account is utilised in several parts of the system. First of all, the user name and organisational prefix is stored together with the persistent identifiers which are created during the workflow. Secondly, the user information also is stored within the data set and query metadata. We decided to allow this redundancy, as the systems for handling the metadata for persistent identification, users and data may be separated into different systems and therefore should remain independent. Consistency checks enforce the data integrity upon data insertion and deletion at all times.

The system also utilises session management for handling user interactions and for separating the actions of all active users. Currently all users may create tables for their data in one shared database schema. The login data is stored in a dedicated database schema for managing the users. The main table describes the user object, which stores an id, user name first name, last name and email address as well as the organisational id which establishes the relationship with the persistent identifier system. Programmatically the user authentication module is a sub-module of the database package. It provides separate API which allows creating, deleting and changing user data such as user names and passwords.

The password is stored encrypted with a Blowfish password hash by using the library `jBCrypt`<sup>20</sup> as described in [53]. This encryption technique is future proof, as it allows to increase the computational power which is needed to decipher the hash by simply adapting the parameters as needed. The code snippet in Listing 4.6 shows how a user is created by the User API. The clear text password supplied during the registration process is hashed by the `hashPassword` method in line 19. By increasing the salt parameter, the number of rounds of the hashing algorithm is increased, which yields a more secure hash value.

---

<sup>20</sup><http://www.mindrot.org/projects/jBCrypt/>

Listing 4.6: Creating a new User

```

1      /*
2      * Create a new user and hash the password by using the jBCrypt library in version
3      * 0.3m.
4      * */
5      public User(String username, String firstname, String lastname, String password,
6                  int organizational_id) {
7          this.username = username;
8          this.firstName = firstname;
9          this.lastName = lastname;
10         this.password = this.hashPassword(password);
11         this.organizational_id = organizational_id;
12     }
13
14     /*
15     * Hash the password
16     * * * */
17     private String hashPassword(String inputPassword) {
18         // Hash the clear text password with the salt
19         // The parameter of the salt is the number of rounds of hashing to apply. This
20         // factor can be increased.
21         String hashedInput = BCrypt.hashpw(inputPassword, BCrypt.gensalt(10));
22         return hashedInput;
23     }

```

Figure 4.8 shows the ER diagram of the user table. The user data is versioned by utilising the Hibernate Envers system, thus changes to a user data are traceable.

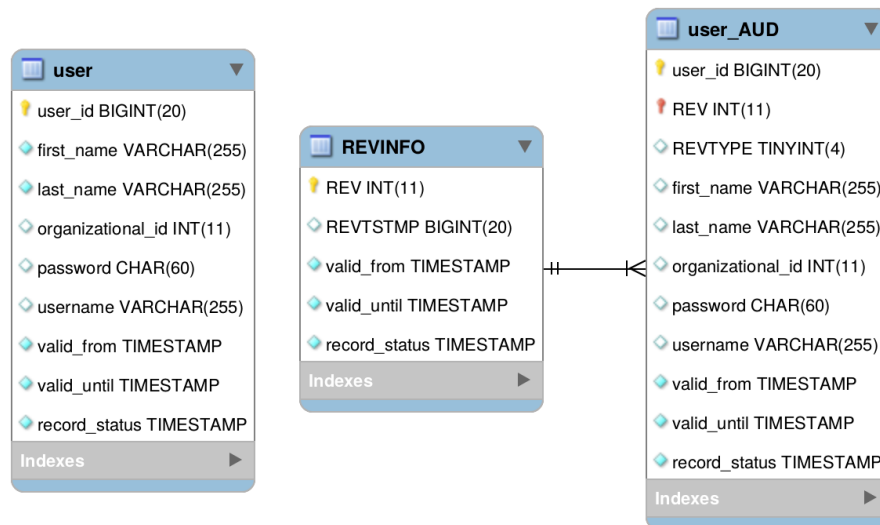


Figure 4.8: ER of the user table

There is currently no authorisation model implemented, which differentiates between user roles. All users may view all tables which are currently available for creating subsets. Future versions could incorporate a more sophisticated user management.

### 4.3.2 The Main Menu

After login in the user is presented with a simple main menu. As depicted in Figure 4.9, the user can chose between uploading a new data set, updating existing data, creating a subset and resolving a PID to the landing page.

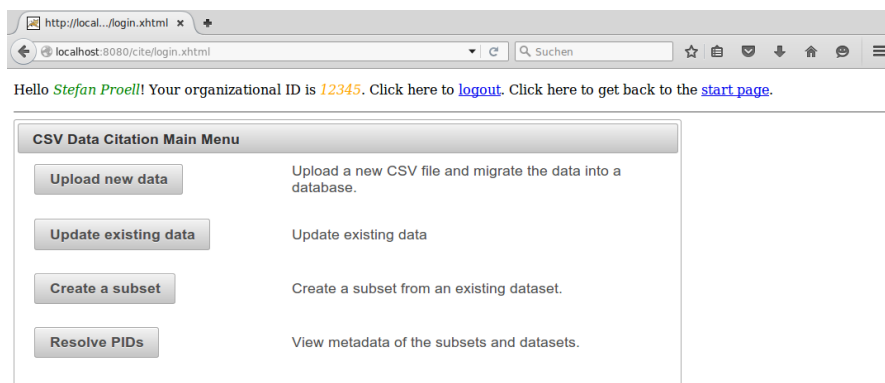


Figure 4.9: Main menu

### 4.3.3 Data Upload

After the user has logged into the system, the user may upload a CSV file. The system checks the extension of the uploaded file and prevents other file types than those ending with the pattern *CSV*. The user needs to enter the metadata of the CSV, that will be uploaded into the system. The interface for the basic metadata is depicted in Figure 4.10.

The system suggests the database schema, where the data will be stored. The author field is prefilled with the user name of the user who uploaded the file. As soon as the user enters a title for the data set, the table name is automatically filled by using the data set title. As MySQL does not allow spaces and other control characters in table names, the system replaces all special characters and applies a simple word stemming algorithm, which removes common short prepositions such as *the* and *a*. The table name is prefixed with *csv\_* and the first name of the user. The user can enter a longer description of the data set, which serves as a textual description of the base table on the landing page.

In a second step, the user uploads the CSV file to the system, by selecting the file in the upload dialog, as shown in figure 4.11.

After submitting the file, the CSV file is analysed and the system reads the headers of the columns. In a third step, the user needs to select the primary key. As described earlier, the primary key is essential for differentiating records from each other, it is unique by definition. The system always provides the possibility of using an artificial primary key, if the data set should not provide a unique key itself. The *ID\_SYSTEM\_SEQUENCE* is an automatically incremented integer field, which is increased for every record read from

## CSV Datafile Upload

### Step 2: Upload the data

Select one file from your hard disk. It will be uploaded to the systems and analyzed. The system currently only supports a single file per upload.

#### Sample datasets

You can download one of the following sample datasets from the links below.

- [Addresses \(1k\) Testdataset](#)
- [World Factbook](#) Adapted from this [source](#)
- [WHO Health DataSet](#) Adapted from this [source](#)
- [Simplified WHO Health DataSet](#) Adapted from this [source](#)
- [MSD 1k subset](#)
- [MSD 10k subset](#)
- [MSD 100k subset](#)

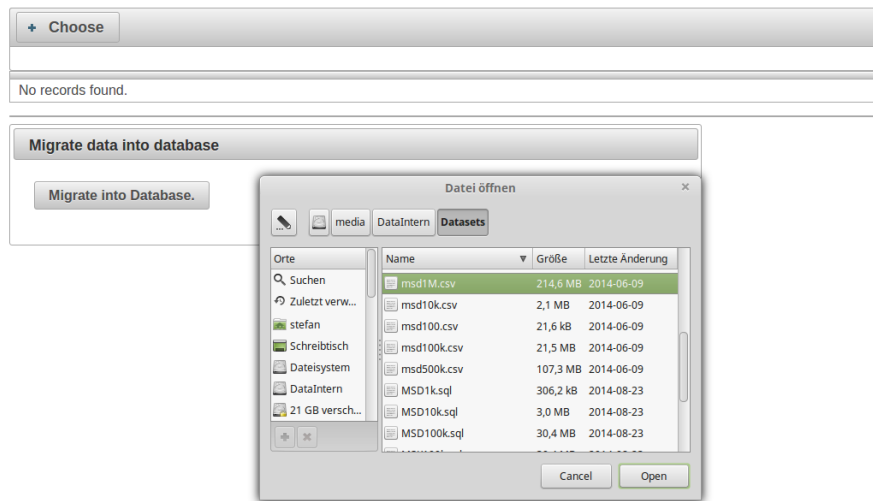


Figure 4.10: Providing the metadata

the file. Figure 4.12 depicts the columns, some of the available columns have been greyed out or removed for increasing the readability. After the user has pressed the *Migrate* button, the system verifies the uniqueness of the key and migrates the CSV data file into the RDBMS.

#### 4.3.4 CSV Migration

If there are headers available, the system cleans the header names by replacing spaces and special characters with dashes. The system estimates the data type of a column by iterating over all records per column and trying to map the value to a set of most common data types. This includes dates, integers, decimals, strings and boolean values. If a column does not have a uniform data type for all its values, the system treats the column as strings (*VARCHAR*). The data type is required for sorting the records either lexicographically or numerically. For defining the length, the system iterates over the CSV file columns and finds the largest value per column.

The system imports the data by using simple *INSERT* statements. The insert



## CSV Datafile Upload

### Step 3: Choose the Primary Key

Be aware that a primary key and also a combination of key forming a compound primary key must be unique. The system automatically creates an internal sequence number column. If you can't provide a unique column as primary key, you can select the insert sequence number as unique key.

**Select Columns**

Primary key

- ID\_SYSTEM\_SEQUENCE
- track\_id
- title
- song\_id
- release
- artist\_id
- artist\_mbid
- artist\_name
- duration
- artist\_familiarity
- artist\_hottness
- year
- digitalid
- audiofile
- lastfm
- numlastfm
- numlastfmmatched
- rpfeatures
- audiofilelength
- simplastfm

Select primary key.

**Migrate data into database**

Migrate into Database.

Figure 4.11: Uploading a file

timestamp is automatically assigned by the RDBMS at the time the data is persisted in the database table. Each new record is marked as *INSERTED* and can be used immediately after the transaction is closed. Once all records of a CSV file have been imported, the system assigns a PID to the base table and presents a clickable link to the landing page of the newly imported data set. Figure 4.13 shows a data set landing page.

The data set landing page provides the PID and the metadata, which was entered by the user during the upload process as described in Section 4.3.3. Additionally the number of active, i.e. non-deleted records, is displayed in the interface. The system provides a simple citation text, which can be copied into reports and documents. Furthermore, users can download the whole data set again as CSV file and re-use it in external applications.

### 4.3.5 Subset Creation

Once the data has been imported into the database system, users can create subsets by using the interface. The Web interface was described in Section 4.1.7. It utilises the JavaScript library jQuery<sup>21</sup> and the DataTables plugin<sup>22</sup> for rendering the data

<sup>21</sup><https://jquery.com/>

<sup>22</sup><https://www.datatables.net/>

## CSV Datafile Upload

### Step 3: Choose the Primary Key

Be aware that a primary key and also a combination of key forming a compound primary key must be unique. The system automatically creates an internal sequence number column. If you can't provide a unique column as primary key, you can select the insert sequence number as unique key.

**Select Columns**

Primary key

- ID\_SYSTEM\_SEQUENCE
- track\_id
- title
- song\_id
- release
- artist\_id
- artist\_mbid
- artist\_name
- duration
- artist\_familiarity
- artist\_hottness
- year
- digitalid
- audiofile
- lastfm
- numlastfm
- numlastfmmatched
- rpfeatures
- audiofilelength
- simplastfm

Select primary key.

**Migrate data into database**

Migrate into Database.

Figure 4.12: Specifying the primary key

dynamically. The DataTables plugin can render tabular data from a JSON source and it allows server side processing. We implemented a Web service which queries the MySQL database for the data in an Ajax fashion, i.e. the data is fetched asynchronously, every time the user interacts with the Web interface. Instead of transmitting potentially large data sets over the network, we implemented server side pagination and the dynamic selection of columns and filter criteria based on the preferences by the user.

Users can enter text based filters for each column and select which column they want to include in their data set. The sequence of columns can be re-arranged with drag and drop, the sorting of each column can be defined as ascending or descending. Figure 4.14 depicts the interface.

Once the user finishes the selection process, the title and a description of the created subset can be provided. As soon as the subset is confirmed, the system approves whether or not the subset and the query respectively have been issued before and assigns a new persistent identifier if necessary. The system responds with a success message including a link to the landing page of the newly created subset, as depicted in Figure 4.15.

Figure 4.16 shows a landing page of the subset. The landing page contains a reference to the parent data set (super set) and its PID. Also it provides a clickable link for the convenience of the users. The citation text is generated automatically and provides

## Dataset Landingpage

[12345/gLmWBKibcx](#)

### Landing Pages - Dataset Metadata

Dataset Title	The Million Song Dataset
Dataset PID	12345/gLmWBKibcx
Dataset upload timestamp	2016-08-17 10:52:07.0
Dataset author	Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere
Description	The Million Song Dataset (ISMIR 2011) is a freely-available collection of audio features and metadata for a million contemporary popular music tracks. The core of the dataset is the feature analysis and metadata for one million songs, provided by
Resolver URL	<a href="http://localhost:8080/cite/dataset-landingpage.xhtml?requestPID=12345/gLmWBKibcx">http://localhost:8080/cite/dataset-landingpage.xhtml?requestPID=12345/gLmWBKibcx</a>
Number of active records in this dataset	1 000 000

---

Suggested citation text: Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere (2016): "The Million Song Dataset", PID [ark:12345/gLmWBKibcx]

### Download area

Download Full DB  Download the full database as CSV file

Figure 4.13: A data set landing page

also the PID to the parent data set. Users can obtain the subset in the version of the execution time of the subsetting process or have access to the most recent version. There is also a link for downloading the parent data set as CSV file. All downloadable data sets are created on demand by re-executing the query.

### 4.3.6 Publish and Reference the Subset

As soon as the subset has been created, the PID becomes resolvable in via the PID service. Users can utilise the assigned PID in their reports, publications and documents for referencing the data set in an unambiguous way. For resolving the data set, users need to know the URL of the resolver service where they can enter the PID, as described in Section 4.3.7.

### 4.3.7 Retrieve Metadata and Subsets

Users may enter the PID into the system or select the appropriate subset from the available data sets by using the menu depicted in Figure 4.17. Users may select the super set from the drop down menu and then chose to display the super set landing page or list the available subsets and access their landing pages. The resolver API offers several methods for accessing the data sets in a programmatic way. Users can resolve existing PIDs and can retrieve lists of available PIDs per organisation. The resolver service is identified with a URL, which is per definition unique. The test deployment is

Hello *Stefan Proell*! Your organizational ID is *12345*. Click here to [logout](#). Click here to get back to the [start page](#).

### Select Dataset

Pick the table which contains the data you are interested in. After clicking on the button *Load table*, you will see the list of columns of this table.

Database schema

Table name

### Data Selection Interface

Show  entries Search:

ID_SYSTEM_SEQUENCE	Country	Area_sq_km	Birth_rate_births_1000_population	Curre
1	Afghanistan	647500	47.02	(Data
2	Akrotiri	123	(Data n/a)	(Data
3	Albania	28748	15.08	-5040
4	Algeria	2381740	17.13	11900
5	American Samoa	199	23.13	(Data
6	Andorra	468	9.00	(Data
7	Angola	1246700	44.64	-3788
8	Anguilla	102	14.26	(Data
9	Antarctica	14000000	(Data n/a)	(Data
10	Antigua and Barbuda	443	17.26	(Data

First Previous 1 2 3 4 5 Next Last

### Create a new subset

Provide a title for the subset:

Provide a dataset description:

Figure 4.14: Create a subset

**i** **ResultSet Hash** The result set has this hash:  
61dfa36bd8cfd249b79c122ef77891ff4fbf9ee5  
**Subset stored** Find details at [Landing page](#) ✕

Figure 4.15: Success message with the hash and link to the landing page

available at the local address <http://localhost:8080/pid/service/resolver/ark:/6789/7WwLdBc7Jvwh>.

The landing page offers a download button, where the user can retrieve the subset as CSV file again. The system re-executes the query, retrieves all required information from the Query Store and creates a CSV file on the fly.

## Subset Landingpage

[12345/qmZi2wO2vv](#)

Landing Pages - Dataset Metadata	
Subset PID	12345/qmZi2wO2vv
Subset author	Stefan Proell
Dataset PID	12345/cLfh9FjxnA
Dataset upload timestamp	2015-10-07 10:47:32.0
Dataset author	CIA
Subset creation timestamp	2015-10-07 10:51:55.0
Subset hash	61dfa36bd8cfd249b79c122ef77891ff4fb9ee5
Subset query hash	aed16bdfde08d86b22fcb87f5d7dee3d5e9ed2ce
Description	Key indicators for Austria
SQL string	<pre>SELECT 'outerGroup':Country,'outerGroup':Area_sq_km,'outerGroup':Birth_rate_birtns_1000_population,'outerGroup':Current_account_balance,'outerGroup':Internet_hosts,'outerGroup':Internet_users,'outerGroup':Population FROM stefan_worldfactbook AS outerGroup INNER JOIN (SELECT Country,max(LAST_UPDATE) AS mostRecent FROM stefan_worldfactbook AS innerSELECT WHERE (innerSELECT.RECORD_STATUS = 'inserted' OR innerSELECT.RECORD_STATUS = 'updated' AND innerSELECT.LAST_UPDATE&lt;="2015-10-07 10:51:55.327") GROUP BY Country) innerGroup ON outerGroup.Country = innerGroup.Country AND outerGroup.LAST_UPDATE = innerGroup.mostRecent WHERE UPPER('outerGroup':Country) LIKE UPPER("%Austria%") ORDER BY 'outerGroup':id' asc,'outerGroup':ID_SYSTEM_SEQUENCE asc,'outerGroup':first_name' desc</pre>
Dataset landing page	<a href="http://localhost:8080/cite/dataset-landingpage.xhtml?requestPID=12345/cLfh9FjxnA">http://localhost:8080/cite/dataset-landingpage.xhtml?requestPID=12345/cLfh9FjxnA</a>
Subset landing page	<a href="http://localhost:8080/cite/subset-landingpage.xhtml?requestPID=1234512345/qmZi2wO2vv">http://localhost:8080/cite/subset-landingpage.xhtml?requestPID=1234512345/qmZi2wO2vv</a>
Suggested citation text:	Stefan Proell (2015) "Austria Facts" created at 2015-10-07 10:51:55.0, PID [ark:12345/qmZi2wO2vv]. Subset of CIA: "The CIA WorldFactbook", PID [ark:12345/cLfh9FjxnA]

Download area	
Download CSV Subset	<input type="button" value="Download"/> Download the CSV data of this subset at the execution time of the query
Download Latest Subset	<input type="button" value="Download"/> Download the CSV data of this subset at its current state
Download Full DB	<input type="button" value="Download"/> Download the full database as CSV file

Figure 4.16: A subset landing page

## 4.4 Addressing the RDA Recommendations

The aim of this work was to validate the RDA recommendation for dynamic data citation by implementing a platform capable of making CSV data citable. The implementation we introduced in Sections 4.1 until 4.3 described this platform in detail. The following subsections provide an overview how the recommendations have been addressed.

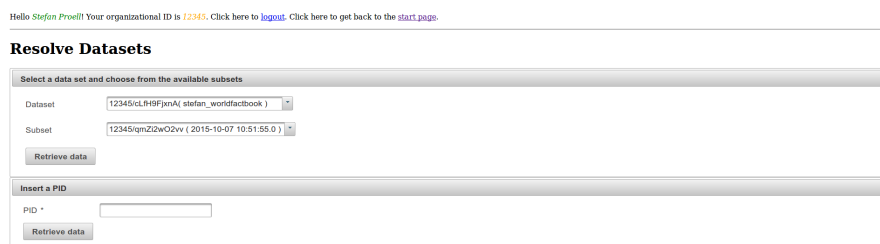


Figure 4.17: The resolver service interface

#### 4.4.1 R1 - Data Versioning

Versioned data is a core requirement for accessing a specific revision of a record at a later point in time. During the data migration process from the CSV files into the system, the system automatically applies a simple versioning scheme, by adding additional metadata columns to the table. The system traces each event which influences a record and marks it with an event type and the exact timestamp. We utilised a single table approach as it was the simplest method and does not require history tables and more complex queries. In addition, we also implemented audit tables for the system information we store internally. This allows us to trace all changes which have been introduced to the metadata of queries, users and persistent identifiers as well.

#### 4.4.2 R2 - Timestamping

Timestamps are needed at several occasions within our implementation. As described above and in Section 3.2.2, timestamps are used for versioning the revisions of single records as they store the exact time and data of a change for the data. The query store also utilises timestamps for persisting the exact execution time of a query. Upon later retrieval, the system retrieves the execution time of a query based on its PID and then re-executes the original query considering only those records, which have been valid at the time of the original execution.

#### 4.4.3 R3 - Query Store

The Query Store is a concept introduced by [50] and a core component of this implementation. We implemented the Query Store by mapping the metadata of a query into a relational database schema and augmenting the query metadata with additional information regarding the timing and users. Queries are generated by the user interface, which allows us intercepting the significant properties of each query. Each query gets a unique PID assigned, which allows referencing the query at a later point in time.

#### 4.4.4 R4 - Query Uniqueness

The system needs to detect duplicate queries, taking into account that each PID references unique object. We compute the uniqueness of a query by using normalised queries (i.e. the system generates the queries in exactly one way) and by sorting the parameters. Parameters can be commutative (i.e. their sequence does not matter, as with conditionals) or they can be non-commutative (such as column sequences). For the former kind of parameters, we re-arrange them in an alphabetical order to detect identical parameters. For the latter parameters, we sort them in their sequence of occurrence. This allows us detecting identical queries and therefore the system can assign existing PIDs to already existing queries.

#### 4.4.5 R5 - Stable Sorting

The sorting of a subset is fundamental for further processing. For this reason we store the sorting columns and their sorting criteria (ascending or descending) as well as their sequence of occurrence in the Query Store. If no sorting is specified, we apply a default sort on the unique identifier (which, in turn, is based on the insert sequence of the records into the system). This allows maintaining the sorting of each result set and subset respectively.

#### 4.4.6 R6 - Result Set Verification

For establishing trust in the correctness of the subsets retrieved, we need to implement a verification mechanism, which allows users to judge the authenticity of the subset. In order to do so, we developed a hashing scheme which allows detecting missing, added or re-arranged (i.e. the sorting changed) records in a result set. Users can re-compute the hash of their data set and compare the hash value with the one stored in the query store.

#### 4.4.7 R7 - Query Timestamping

As described earlier, we store the execution time of each query as a timestamp in the Query Store. The timestamp of the query execution is determined by the DBMS, when a new record is inserted into the Query Store table. For this reason we can assign an exact timestamp for each query, the database automatically stores the timestamp in the table upon the creation of a new query. This timestamp is then used for referencing the valid records in the database tables. Users may create subsets of a data set which was updated, by re-executing an existing query against the changed data. In this scenario, timestamps are used for detecting valid records in the database.

#### 4.4.8 R8 - Query PID

We assign custom, ARK [37] style PIDs to each query which generated a subset of data. For creating the PIDs, we implemented a Persistent Identifier Service which can generate various kinds of identifiers and ensures their uniqueness within one instance by utilising

database concepts such as primary keys. Before a query gets a new PID assigned, the system checks whether or not the query is unique and therefore new. If the query is unique, the system assigns a new and likewise unique PID. If the query has been issued before, the system verifies, whether or not the data within the result set has changed. This can be detected by the timestamp of the records. If a newer timestamp than the query execution time exists in the database, then the data was updated in the mean time. In this case, the query gets a new PID assigned. If the data is unchanged, the system does not create a new record in the system for this query, but responds with the PID of the identical query.

#### **4.4.9 R9 - Store the Query**

The Query Store is the essential entity in this implementation and responsible for storing all information required for re-executing a query again in the future in order to obtain the same results. We create and manage the Query Store data objects with Hibernate and ensure that all operations are performed within one transaction. All Query Store data is stored within audited tables which allow tracing changes to the data if they become necessary. The metadata stored within the Query Store can be exported and may outlive the source data. The data allows understanding how a subset was created, even if the underlying technology changed or becomes unavailable.

#### **4.4.10 R10 - Citation Text**

Providing an citation text decreases the burden of researchers for citing the data which they used. The citation text can be copied from the landing page and pasted into any kinds of documents. We implemented a simple system which utilises the information stored in the Query Store and concatenates the bibliographically important metadata as a plain text statement. It includes author name, creation data, data set title and the PID of the database table which contains the source data. In the case of a subset, the citation text also includes a reference to the parent data set. The same information could also be provided as BibTex record.

#### **4.4.11 R11 - Landing Page**

The landing page is important for the interaction with users. It offers a visualisation of the data stored in the Query Store and allows viewing details about data sets and their subsets for human consumption. The landing page is dynamically created JSP page and retrieves the data based on the provided PID. The resolver service points to the landing page and is the point of reference for all data sets. It also allows to download the data set again as CSV file for further processing and provides the citation text which can be copied by users.



#### **4.4.12 R12 - Machine Actionability**

Although the aim of this work was to facilitate the data citation process for human users, the system also provides a large variety of APIs which allow the programmatic access to the services. The Persistent Identifier Module offers JSON exposure of the metadata of the queries and allows to search and list PIDs based on organisation prefixes and names. The Query Store Module offers a comprehensive API for creating, managing and retrieving queries and their associated metadata. It was designed in a way that it can be used in other settings than SQL and stores the metadata of each query in an abstract way. The Verification Module also allows computing hashes from inputs served via the API and can be plugged into other components if required.

#### **4.4.13 R13 - Technology Migration**

We decided to utilise components which are widely used in the area of academic prototype implementation and refrained from using exotic libraries and specialised components. As far as possible, we only utilised well documented and open source software components for our work. All the metadata is stored in a MySQL RDBMS instance. As MySQL is a widely used technology, several migration tools exist, which allow exporting the data into other systems. Also no internal MySQL specific database objects, such as stored procedures or triggers are used. Thus the data can easily be exported into simple, text based formats which either allow porting the system into a new data representation or even print the data and process it manually, if necessary. The components have been designed with extensionability and modularity in mind, allowing to replace single functions by newer implementations if necessary. We chose simplicity over complexity and implemented the methods required for making data citable in a way they can be easily reproduced in other technological settings. The original SQL statement of the queries are stored, but only used for documentation purposes. The queries are stored in an abstract fashion, allowing to migrate them into other query languages by simply translating the parameters used for each query.

#### **4.4.14 R14 - Migration Verification**

Having future technological changes in mind, it is essential to provide means how future implementations may verify, that the result sets which they produce after the migration into a new technological environment still are valid. For this reason we implemented a simple hashing scheme for both, the queries and the data, which does not rely on the query language or data representation. For queries, the hashing scheme only utilises the parameters, which define the subset and their sequence, where applicable. For the result sets, we only utilise the PID, sequence of columns and the sequence of records as input for the hash function. For this reason the hashes can be recomputed even with very simple means. If the hashing algorithm changes, for instance because it has been

broken<sup>23</sup>, our implementation allows replacing the hash values by a new hashing function and recomputing the old and the new values for the verification at this step.

#### 4.4.15 Summary

The RDA Recommendations for Making Research Data Citable provide a basic set of requirements which need to be fulfilled for implementing a citation system. The RDA guidelines offer a collection of principles, which define how data can be rendered citable by focusing on queries instead of data exports. The guidelines are concise and do not intend to suggest concrete implementation suggestions, but only the fundamental concepts. In our work, we successfully implemented a system for making CSV data citable by following the RDA guidelines and implemented all of the available recommendations<sup>24</sup>.

---

<sup>23</sup>An overview of hash functions and their life times can be found here <http://valerieaurora.org/hash.html>

<sup>24</sup>Based on the version from 09 June 2015 <https://rd-alliance.org/rda-wgdc-revised-version-recommendations-making-data-citeable-published.html>

# Evaluation

## 5.1 Overview of the Evaluation Process

In the previous sections, we described the prototype and its implementation for making CSV data citable. In this section we present an evaluation of the system, by comparing it to a comparative system.

### 5.1.1 The Evaluation System: Git and CSV2JDBC

In order to evaluate the prototype system (PS), we need to define an evaluation system (ES). As we described in Section 3.1.4, researchers often handle versioned data by creating copies of their data sets. Handling multiple copies of data is a cumbersome and error prone task we wish to overcome with approaches that we presented in our solution in Section 4.1. In order to evaluate the performance and applicability of our approach, we need to compare our implementation with a reference system. As source code management systems are commonly used in software development and increasingly find users in the domain of research data [54], we decided to implement the evaluation system based on Git. Git is used for tracing changes in files and allow users accessing previous versions of their documents later. It has been designed for handling source code, which is usually available in its textual representation. Although Git can handle binary data as well, its more powerful features can only be applied for textual data.

This makes Git an ideal candidate for handling CSV data and for storing updates to a specific CSV data set. The software was not designed for the concept of subsetting and therefore it does not provide any tools for creating a subset from a CSV data file.

#### A Basic Workflow for Git

The authors of [49] describe three methods for data citation, two of which are based on Git. The first method covers the implementation of this thesis, which utilises relational

databases and the Query Store. The second method utilises Git for storing the CSV data in a versioned fashion. It allows to execute versioned scripts on the versioned data. This scripts are used for retrieving a specific subset of the data, for instance in the language R<sup>1</sup>. We adapted this approach in this thesis, by creating the subsets with a database driver which can read CSV files directly, without the overhead of an actual database engine. The required details about subsets are stored in and retrieved from the Query Store. This method is the evaluation system used in this thesis, as it also utilises the same Query Store infrastructure. The third method described in the previously mentioned work also uses Git for data versioning and storage, but implements a dedicated Query Store which is not compatible with the solution presented in this work. For this reason we decided to utilise the approach based on Git and the SQL language for generating subsets.

The process for creating versioned subsets with Git is depicted in Figure 5.1. It

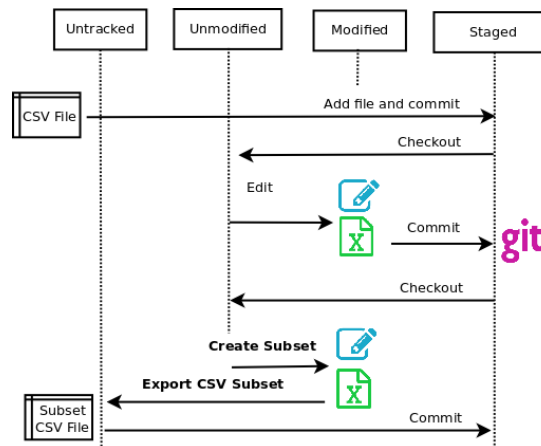


Figure 5.1: CSV workflow with Git

starts with an unversioned CSV data file. The user needs to add a data file into the version control of Git (*add*) and then needs to make the file known to the system with a (*commit*). The file is now in the staging area monitored by Git. In order to create a subset of a CSV file and allow to access a specific version of the data set, users need to utilise an external tool such as Excel. Thus the user retrieves the current version of the file (*checkout*) and creates a specific subset by using the external tool. Then, the user saves the newly created subset as a new CSV file which is currently not tracked by Git. After adding the CSV subset to Git, the user can retrieve this specific version of the subset by checking out the appropriate file. The user can also edit the parent CSV file and store the changes with Git. Other users can retrieve each updated version from the Git repository, but there does not exist a possibility of updating the subsets, without repeating the subsetting process manually.

<sup>1</sup>R Project for Statistical Computing [www.r-project.org](http://www.r-project.org)

## Introducing Query Based Data Citation for Git

For the evaluation of our prototype, we need to compare our implementation with a system meeting the same basic principles. For this reason, we need to get rid of the manual subset creation process (by using Excel or any other application) and use a query based approach for retrieving the subsets from the CSV file.

Thus we utilise the *jdbc-driver-csv* implementation of the CSV2JDBC<sup>2</sup> library, which provides a read only interface for interacting with CSV files. The driver is available as open source software and can be integrated as a library. It supports the following grammar for SELECT statements, as shown in Listing 5.1.

Listing 5.1: The SELECT Grammar of the jdbc-driver-csv Driver

```

1 SELECT [DISTINCT] [table-alias.] column [[AS] alias], ...
2 FROM table [[AS] table-alias]
3 WHERE [NOT] condition [AND | OR condition] ...
4 GROUP BY column ... [HAVING condition ...]
5 ORDER BY column [ASC | DESC] ...
6 LIMIT n [OFFSET n]

```

The JDBC driver for CSV files offers the basic functionality that we need for retrieving subsets from CSV files with SQL. The integration of this driver into the workflow is depicted in Figure 5.2

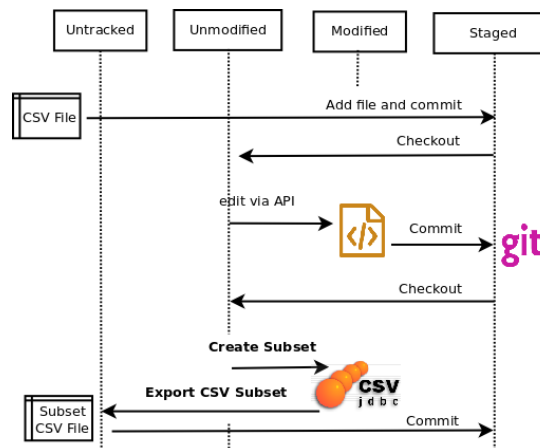


Figure 5.2: Git workflow with JDBC

Instead of storing each created subset as a new CSV file, which remains detached from its parent CSV data set in Git, we apply a simple query mechanism on the versioned parent data file. In this scenario, users can edit the CSV data file programmatically, by adding new records as a new line in the CSV file. Records are deleted by simply removing the corresponding line from the file. Updates alter the content of a record, where each cell of the CSV file can be altered except the primary key column. After each INSERT, UPDATE or DELETE operation, the user needs to commit the change to the Git repository. Each commit defines the granularity of the data set, hence if the user

<sup>2</sup><http://csvjdbc.sourceforge.net/>

commits after each change, every change can be traced. If the user decides to update or change the file in batch, the system may resolve changes on batch level.

For retrieving a specific version of a subset, the user needs to checkout the revision of interest and re-execute the original query against the checked out revision of the file. In this scenario, the CSV files do not need to store the timestamp as a new column, because Git takes care of the timestamping.

Creating a subset does not lead to a new file in the Git repository, but only to a data export based on the SQL query the user issued. Obviously, in this scenario the Query Store could be used for persisting the information about the query as well.

### 5.1.2 Method

In order to compare the two approaches, we apply the evaluation methods described in this section. The evaluation of the systems is CSV centric, all operations are performed on the CSV file directly. Subsequently, the new information contained in the CSV file is propagated in both systems. The evaluation is based on the following steps:

1. Create CSV test data sets
2. Ingest the data into the Git and the SQL system
3. Perform INSERT, UPDATE, DELETE operations on the CSV data to simulate the evolution of the files in both systems
4. Upload the changed data set to both systems
5. Create subsets on the versioned data and export the data sets

We measure the required execution time for the SELECT, INSERT, UPDATE and DELETE operations and compute the storage demand for both systems.

#### The Test Data Generator and the Generated Test Data Sets

For being able to compare both systems, we need to utilise test data sets, which fulfil defined criteria. Each CSV test data file consists of a specified number of columns and number of rows. The first row of each CSV file contains the column names, which are generic and follow the naming pattern *COLUMN\_1*, *COLUMN\_2*, etc. A record is one row in the file, where each cell (or field) of a record is a random, alphanumeric string. For evaluating the prototypes, we decided to utilise strings as the values for all columns. This prevents the database system of utilising any data type specific optimisation and allows to compare the performance in both systems independently of the data type of the columns. The length of each string can be defined as well as the variance of that length. Figure 5.3 shows an example of a generated CSV file.

The first column of each data set serves as the primary key column. The test data generation tool developed for this evaluation ensures that the values in the first column are unique and can be used as primary key automatically. The length of the strings in the

COLUMN_1	COLUMN_2
QUOVMJZH0EX8IWPI	ZLXS0EJAIY6IN9DAZRA
E6QUBPY8M0EJDHDQYMTTI	RFILEHBFJT4TZHB
XZBBLPJBJJBJWJA	09Y1M2YW0MBN7I
TGKGQQLJJLGE8F8JHU7TRE	PRMKUJ2MXYKPI
SMU6F37GZZH9F17T9NGAFU	5UDNNDDTVSZJ8EIJ
0QPMJDHGHCWW	2ZTKZDN1YCMQFP
PAYIOYWGNCJQD0	GKPQERIR
QZKHKQDEDCOMGTG	BPS0YJM
SAGG8BS9WPBJ5GQ	RMFIBFET6H
0DU2JL0SJJH1J0	HC4LEVUQEADT
2JJAGZSX0QYV	H04MWKI3TAMGK
2RN6TDN	BDVEFY00QZK
CBXD0PVB1V0R1	TDRCESLEXKAX1KV
PHTRMJNBVBPHXP	SL4
...	

Figure 5.3: Example of a random CSV file

Complexity	Number of Columns	Number of Records	Average Record Length
Simple (SMP)	5	1000	10
Medium (MED)	25	10000	25
Large (LRG)	50	100000	50

Table 5.1: The test data sets

first columns must be large enough to allow to create an appropriate number of unique values. The data generator tool creates randomised data sets of different complexity, as depicted in Figure 5.4. For investigating on the effects of data set complexity, we created three different data sets.

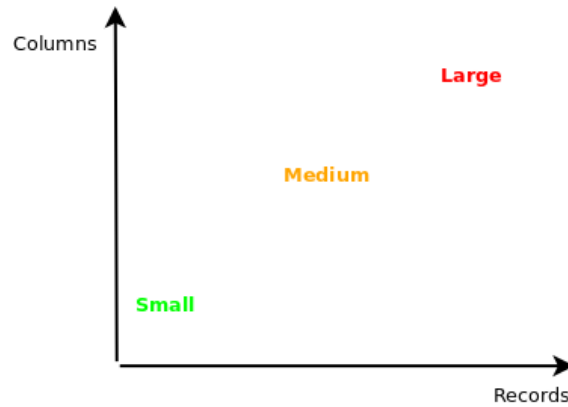


Figure 5.4: Test data set sizes

Table 5.1 shows the specifications of the data sets, which serve as the start configuration for our test runs. The size of a CSV data set is a factor influenced by the amount of columns, the number of records and the average length of the strings per column

## The Test-Runs

A test run creates several CSV data sets based on the properties defined in the previous section. The evaluation system first creates the CSV files and stores them on the disk. Whenever a new CSV file is created, the system requests a new PID from the PID module and assigns the alphanumeric PID as file name for the CSV file.

In a second step, the system iterates over all of the files and inserts them into the Git and the SQL system. In the case of the Git System, the evaluation module creates a new Git repository and adds all data sets into the repository. For the SQL system, we utilise the API in order to import the data into the database. During this step, the system automatically defines the primary key column, adds the versioning information columns and adds indices for performance enhancement.

In the third step, the evaluation system creates query objects and generates random queries based on parameters in an automated way. Each query is executed within both systems and the execution time and storage demand of the operations and the two infrastructures are measured. Details about the test suite are given in Section 5.1.3.

Data sets and the operations applied on them differ largely in the frequency of changes and in the types of changes which occur. Creating subsets is the most common operation for data sets, which remain stable. Inserting new records is the most typical operation for streaming data and log files, where subsets are only created during a system audit, which is likely to be a rare event. Updates and deletes are common operations for data sets where many people collaborate and which are less stable.

It is assumed that growing and changing data sets have an effect on the performance of the system, as the queries are being executed against a larger or different data source at each time. In order to evaluate the effect of the operation type on the data set, we developed a set of test runs applying different scenarios. We evaluate four different scenarios for each data set based on the distribution of the likelihood that a given operation occurs. Table 5.2 shows the scenarios and their distributions of operations.

	SELECT	INSERT	UPDATE	DELETE
S1: Stable Data	1	0	0	0
S2: Infrequent Updates	0.8	0.05	0.15	0
S3: Streaming	0.01	0.99	0	0
S4: Ephemeral	0.1	0.3	0.3	0.3

Table 5.2: The distribution of operations

Depending on the query capabilities of the system, it is assumed that the complexity of the query has an effect on the execution time of the query. For this reason, we introduce



	Q-ESY	Q-STD	Q-CPX
Distribution	0.6	0.3	0.1

Table 5.3: Query distribution

three different kinds of queries. Listing 5.2 shows a simple query, which retrieves one column and applies one filter.

Listing 5.2: Easy Query (Q-ESY)

```
1 SELECT col1
2 FROM table
3 WHERE col1="\%value\%"
```

Listing 5.3 shows a standard query, which retrieves three random columns and applies a filter for three random strings.

Listing 5.3: Standard Query (Q-STD)

```
1 SELECT col4 , col11 , col14
2 FROM table
3 WHERE col4 LIKE "\%e4w\%"
4 AND col1 LIKE "\%tI\%"
5 AND col2 LIKE "\%q2w\%"
```

Listing 5.4 shows a complex query, which retrieves all columns, filters for three random strings and sorts three random columns in a random way.

Listing 5.4: Complex Query (Q-CPX)

```
1 SELECT col1 , col2 , col3 , col4 , col5
2 FROM table
3 WHERE col2 LIKE "\%FPz\%"
4 AND col5 LIKE "\%a1\%"
5 AND col1 LIKE "\%sf\%"
6 ORDER BY col7 DESC, col17 ASC, col2 DESC
```

These queries are randomly generated based on the distribution shown in Table 5.3. The strings used for filtering are also generated randomly and have a length of 2 and a variance of 1 characters. For this reason the result sets also vary in their sizes, as the records included depend on the random filtering of these short strings.

A further parameter for measuring the performance of the approach is the amount of operations. Depending on the proportion of INSERTS and DELETES, the data set grows or shrinks. We measure this effect by applying different amounts of iterations for each test run, as depicted in Table 5.4.

We provide test runs for three different size categories (SMP, MED and LRG) of data sets in four scenarios (S1,S2,S3 and S4). We ran 100 respectively 1000 operations on the data sets for all of these combinations, except the CPX data sets, where we only tested 100 operations per run. The reason was that the query time for an non-indexed data set

Test run class	Number of Operations
Short (R100)	100
Medium (R1000)	1000

Table 5.4: Iterations of operations

in the PS exceeded a feasible amount. In total, we evaluated both systems in 20 different configurations.

### 5.1.3 The Test Suite

We implemented a test suite in order to evaluate the systems against each other. After the data has been imported, we randomly generate operations based on the distribution described in Section 5.1.2. We measure execution time and storage demand for SELECT, INSERT, UPDATE and DELETE operations and store this information in a dedicated database denoted *EvaluationDB*, Figure 5.5 shows the ER diagram of this system.

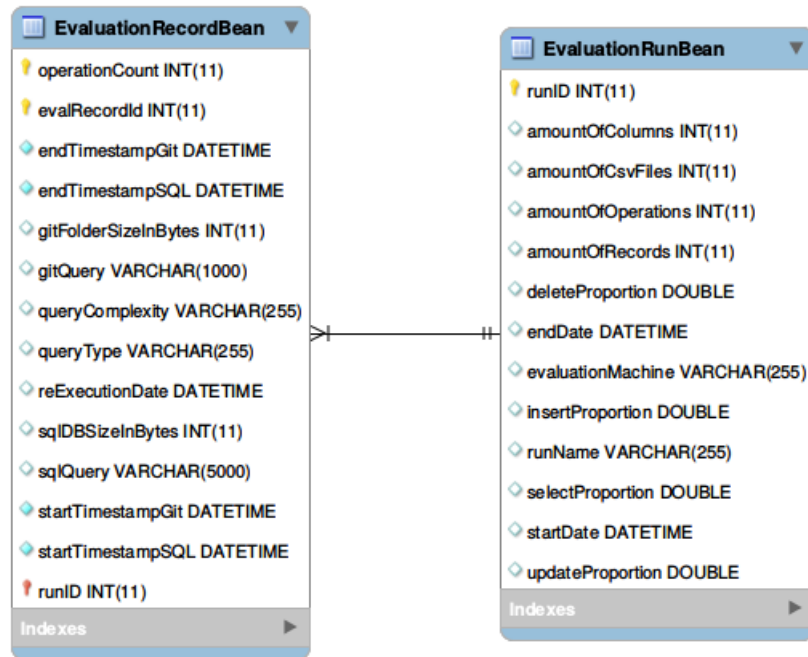


Figure 5.5: Evaluation database tables

### SELECT Statements

Both evaluated systems provide the capability of issuing SELECT statements against the data source. In the case of the ES, the system needs to checkout the most recent revision

of the CSV file first, before it can execute the query. For obtaining a subset in the PS, the system needs to rewrite the query and retrieve the subset from the versioned data.

### **INSERT Statements**

In order to insert new data, we generate  $n$  random strings, where  $n$  is the number of columns in the data set, the API creates a new record for the data set. The test suite API appends a new row to the most recent file from the Git repository and commits the changes into the repository. The updated file is then automatically uploaded into the PS where the new record is detected and inserted into the database table. The integrity of the primary key constraint is enforced by inserting a unique record in both systems.

### **UPDATE Statements**

In order to simulate record updates, we randomly alter a record from the latest version of the CSV file from the git repository. We replace existing values except the primary key with new random strings and commit the changes to the ES repository. We then upload the changed file into the PS, which needs to detect the change and then mark the old version of that record as updated and insert the new data accordingly.

### **DELETE Statements**

The DELETE statement is implemented straight forward. We randomly remove one record from the latest Git version of a file and commit the changes to the ES. We then again upload the altered file to the PS, which needs to detect the missing record and mark it as deleted in the system without actually deleting it.

### **Statements as Transactions**

In order to compare each operation in both systems, we define each operation as one transaction. This entails that statements which require several steps, for instance the UPDATE operation for the ES, are executed within one transaction. Thus the smallest unit of measurement is one single transaction.

#### **5.1.4 Measurements**

The main objective of the evaluation is to measure two factors: time and size. The measurements are persisted in a MySQL database schema, which contains the measurement data for each of the operations. This includes the type (SELECT, INSERT, UPDATE, DELETE), start time, end time and the storage demand in bytes for both systems. Each measurement belongs to one test run, which is identified by a unique id. For comparing the behaviour of both approaches, we store the metadata of all measurements, which can then be plotted as a graph based on the collected data.

### 5.1.5 Measuring Execution Time

In order to measure execution times for all operations, we store the start and the end time at microsecond level. The used version of MySQL 5.7 supports fractional timestamps and allows measuring time with very fine granularity. The time measurement only covers the actual operation and does not include other operations, for instance the creation of Java objects beforehand.

### 5.1.6 Measuring Storage Size

For measuring size we consider the actual data usage per table and its index size, which is used for increasing the performance of the system. MySQL offers in the current version low level performance data, which enables us to measure the storage demand of databases in near real time. The SQL statement for computing these sizes in MySQL is given in Listing 5.5.

Listing 5.5: Retrieve the Data Size From the Table

```
1
2 -- get data and index size per database
3 SELECT SUM(FILE_SIZE)
4 FROM INFORMATION_SCHEMA.INNO_DB_SYS_TABLESPACES
5 WHERE NAME LIKE "CitationDB%"
```

For calculating the size of the Git repository, we use the standard API from Java to retrieve the size of the repository on disk. The following code 5.6 is used.

Listing 5.6: Retrieve the Data Size From the Table

```
1 public int getFileFolderSize(File dir) {
2     int size = 0;
3     if (dir.isDirectory()) {
4         for (File file : dir.listFiles()) {
5             if (file.isFile()) {
6                 size += file.length();
7             } else {
8                 size += getFileFolderSize(file);
9             }
10        }
11    } else if (dir.isFile()) {
12        size += dir.length();
13    }
14    return size;
15 }
```

### 5.1.7 Test Runs

One test run consists of a number of INSERT, UPDATE and DELETE operations, which have been defined in Section 5.1.2.

1. Generate CSV files based on the distribution
2. Select operation according to the distribution defined in Table 5.2. In the case of a SELECT statement, randomly select the query complexity based on Table 5.3.
3. Start timer

4. Perform operation
5. Stop timer
6. Persist measurement data

In order to ensure comparability, the queries in the PS are executed directly against the RDBMS and do not utilise the visual interface for human users. This allows automating the experiments and collect measurement data automatically. For avoiding effects from internal caching, as it is used for instance in MySQL, we clear the caches after each operation. The execution time is measured in micro seconds and the size is measured in bytes. Both measurements are persisted after each execution, the process duration of storing measurement data is not included in the measurements.

### 5.1.8 CSV Exports as Baseline Comparison

In this thesis, we compare the storage demand and temporal requirements between a Git repository and the MySQL based implementation. Additionally we measure the storage demand for each individual CSV subset, in order to compare the overhead needed by the two approaches in comparison to the traditional approach of storing each data set individually. For this reason, we measure the cumulative storage demand of both systems.

### 5.1.9 Limitations of the Evaluation Method

In order to evaluate the prototype implementation, we developed a second system based on the Git source code versioning system. Although we were able to utilise the same language for retrieving the subsets, some of the features could not be implemented in both systems. This includes for instance the integrity enforcement of the data which we use for storing the CSV data in the MySQL server instance, the data type detection, the primary key enforcement and other database specific features. Obviously some of these features have a significant impact on the execution time of the SQL system if they are lacking or not properly set. In a real world scenario, many of these features are used to improve performance, but they require manual tuning and settings which have not been considered in this evaluation. This is the reason why some operations consume considerable more time than they would in real world settings.

Another limitation regards the granularity of Git timestamps. Git has not been developed for supporting high frequency commits of a granularity below one second. Git is not considered a transactional database replacement. For this reason we slowed down the operation execute to one operation per second in order to reflect the capabilities of the evaluation system. This was necessary in order to verify that the queries which are executed in both systems actually produce the same results, as otherwise the mapping of the execution timestamps was not feasible as Git caps off timing information below one second granularity. In this thesis, we focused on the effect of the operations on the storage demand and execution time and therefore executed each operation individually. This is the reason why we did not evaluate more than 1000 operations for most runs, as

this would exceed execution times for the evaluation. Real world scenarios utilising CSV would rather execute batch commands, where multiple records are added, modified or deleted at once, than committing each operation individually.

### 5.1.10 The Evaluation System

In order to establish a comparable execution environment for the test runs, we deployed two virtual machines on a host system. The host system is running Mint Linux 17 on a Intel i5-4570 CPU with 4 cores and 16GB DDR3 memory. Each of the Virtual machines is running Mint Linux 18 on one core with 4 GB memory.

Both systems were executing the test suite and writing the data generated during a test run on their local disk and a local MySQL instance on each machine. The evaluation results have been written into a MySQL server instance on the host machine, where the data has been transferred over a virtual network connection. As each machine only required to transfer one record per operation (and therefore second) to the host system, this resource consumption can be neglected. Collecting the measurement data in a central repository allowed us generating the plots with scripts in a semi-automated fashion.

## 5.2 Results

Table 5.5 shows the specifications of the evaluation runs including the data set size, the query type distribution and the number of operations.

### 5.2.1 Evaluation S1

The graphs in Figure 5.6 and Figure 5.7 for SMP-S1-R100 show a static scenario, where the data is uploaded once and no records are added, modified or deleted. As expected, the storage demand is constant for both systems and we can see in the plots that the storage demand is slightly larger for the PS. The reason for the increased storage demand shown in Figure 5.6 in this scenario for the SQL based approach is the overhead for indices on the primary key, table metadata and other data generated by the RDBMS.

The execution time behaves in the opposite, meaning that the query execution is faster with MySQL than with the Git based approach. As shown in Figure 5.7, the MySQL approach requires less storage time. Running the same scenario with 1000 operations demonstrates the same pattern of a rather constant storage demand for both systems, where the PS requires more disk space for storing the data. again , the execution time is beneficial for the SQL approach. Both evaluation results are shown in Figure 5.8.

Running the second evaluation MED-S1-R100 respectively MED-S1-R1000 with larger CSV files leads to a similar behaviour, as depicted in Figure A.1 and Figure A.2. The large data set scenario is depicted in Figure A.3, it follows the same pattern. All graphs are available in the Appendix A. The test files in the complex scenario have a size of 100 MB in average, which is beyond the capabilities of most editors. As we generated queries

Test Run	Complexity	Distribution	Operations
SMP-S1-R100	SMP	S1	100
SMP-S1-R1000	SMP	S1	1000
MED-S1-R100	MED	S1	100
MED-S1-R1000	MED	S1	1000
CPX-S1-R100	CPX	S1	100
SMP-S2-R100	SMP	S2	100
SMP-S2-R1000	SMP	S2	1000
MED-S2-R100	MED	S2	100
MED-S2-R1000	MED	S2	1000
CPX-S2-R100	CPX	S1	100
SMP-S3-R100	SMP	S3	100
SMP-S3-R1000	SMP	S3	1000
MED-S3-R100	MED	S3	100
MED-S3-R1000	MED	S3	1000
CPX-S3-R100	CPX	S1	100
SMP-S4-R100	SMP	S4	100
SMP-S4-R1000	SMP	S4	1000
MED-S4-R100	MED	S4	100
MED-S4-R1000	MED	S4	1000
CPX-S4-R100	CPX	S1	100

Table 5.5: Evaluation overview

Complexity	Size PS (MB)	Size ES (MB)	Time PS (sec)	Time ES (sec)
Simple (SMP-100)	0.66	0.04	1.49	2.89
Simple (SMP-1000)	0.67	0.04	16.29	17.4
Medium (MED-100)	30	4.2	9.9	46.16
Medium (MED-1000)	30	4.2	109.65	415.20
Large (LRG-100)	408.0	82.16	247.15	641.29

Table 5.6: S1 Evaluation Results

which do not allow the usage of an index for the large table, we can see the increasing execution time. All measurement values for the first scenario are shown in Table 5.6.

### 5.2.2 Evaluation - S2

The graphs in the Figure 5.9 show infrequent updates, where most of the data is read but new data is added and updates occur from time to time. We can see in SMP-S2-R100 that with an increasing amount of records, the execution time of the SQL system is

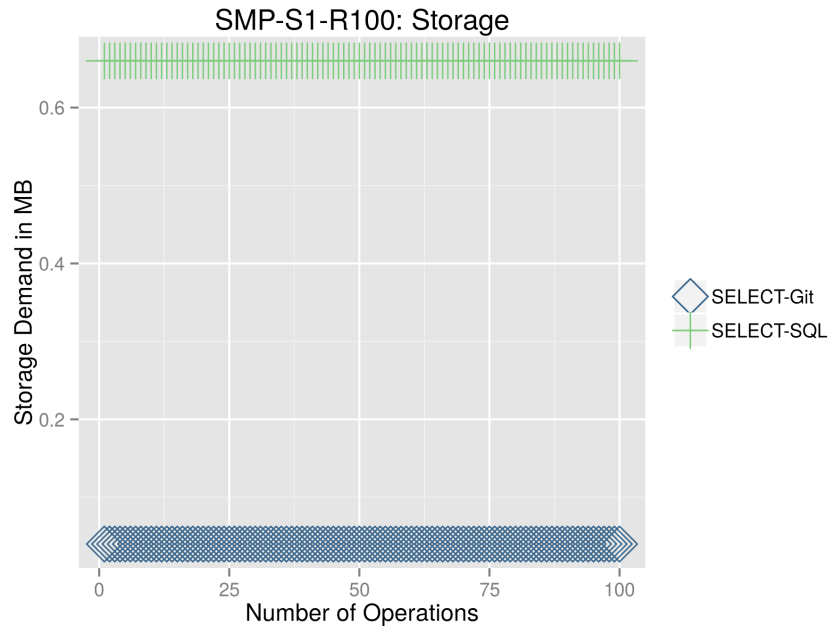


Figure 5.6: Evaluation of the storage demand of SMP-S1

increasing linearly. This is due to the fact that the PS needs to iterate over the existing records and check if the records already exist.

We can see the linearly increasing storage demand for the Git system in comparison with the almost constant storage demand for the SQL system. The behaviour is the same with the medium data sets, as depicted in Figure A.4, which is shown in the Appendix A. This based on the fact that Git stores snapshots of the files that changed for each commit. In its default setting, Git stores for each snapshot a copy of the file if it changed. As in this scenario, the file is changed with every commit, the storage demand is increasing linearly. Details in this issue are given in Section 5.3.1.

Figure A.5 in the Appendix A shows the effect of the update statements nicely for the PS. Each update requires to iterate over the whole data set and check if a record exists. If it exists, the system checks if it has been changed. This is the reason for the vertical gaps in the graph, which do not occur in the simpler ES.

### 5.2.3 Evaluation S3

The graphs in the Figure 5.10 show the results of a streaming pattern, where data is mainly inserted and no data is changed. In this scenario, we can see that the storage demand increases for the ES much faster and is in general higher than the PS. As mentioned earlier the reason for this behaviour lies in the way how Git maintains the storage. In its automatic setting, Git stores for each file which changed between to



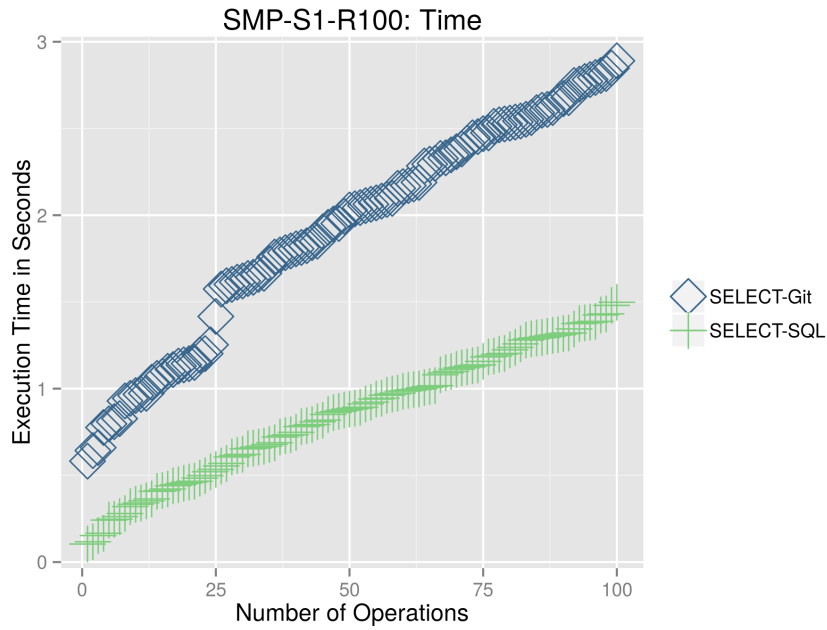


Figure 5.7: Evaluation of the execution time of SMP-S1

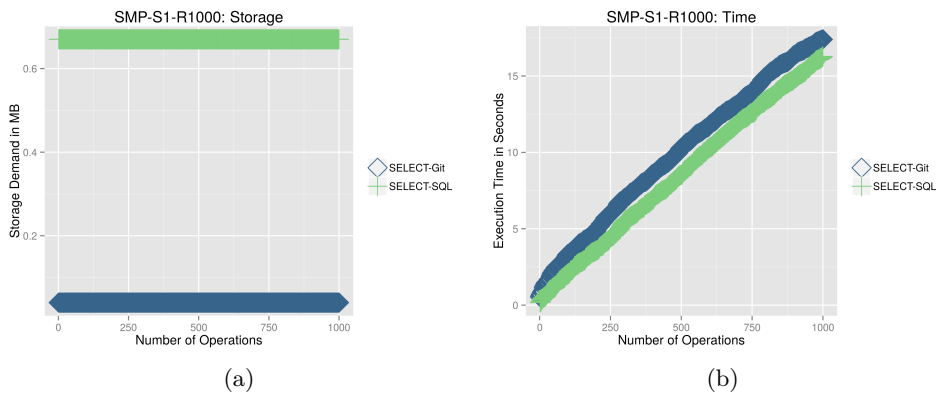


Figure 5.8: Evaluation SMP-S1-1000

commits a new object in the Git repository metadata directory. Details how to improve the storage management of Git are given in Section 5.3.1.

While inserting is cheap in terms for time for the ES, the PS shows a pattern that requires an over proportional time for inserting data with an increasing amount of INSERTS.

The average case depicted in Figure A.6 and the large case in Figure A.7. Both are in the Appendix A and again follow the linear pattern, where the evaluation system

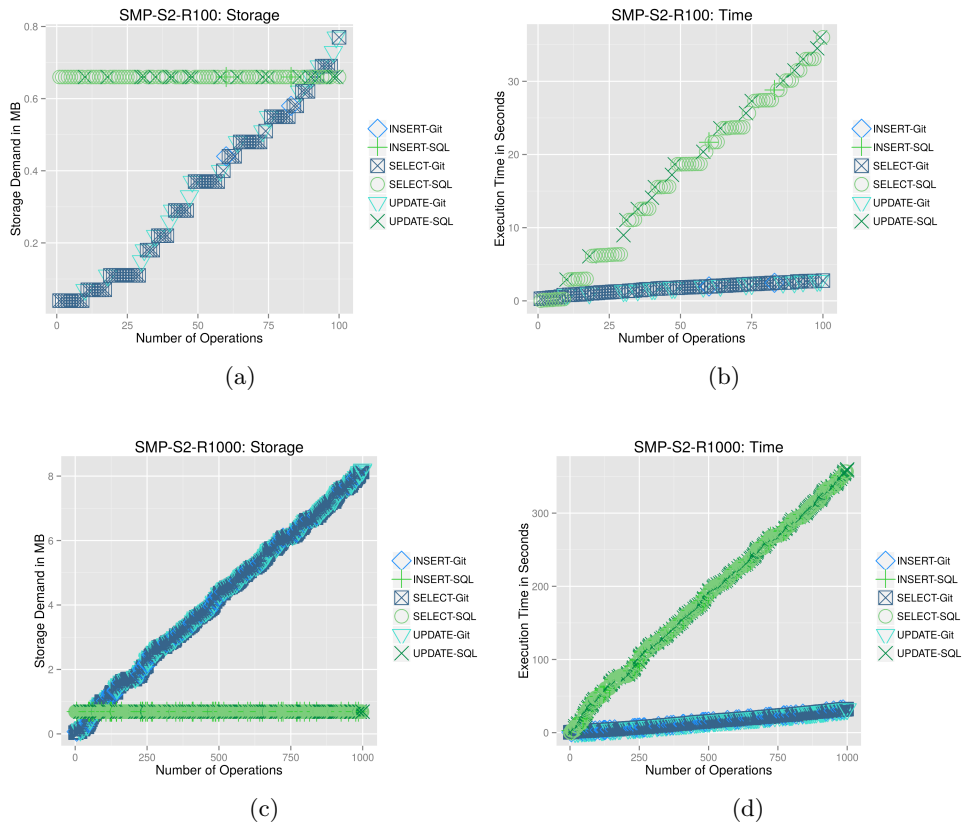


Figure 5.9: Evaluation SMP-S2

manages to insert roughly 100 records per 1000 seconds. This clearly depicts the required overhead if no optimisation are being made at all. Usually the scenario is known to the operators of such a system and therefore the system can be tuned, by using indices on the significant columns, for instance.

The complex scenario which runs a query not allowing indices on a data set with 25 columns and an average string length of 25 characters put the evaluation system to stress, as for each insert, the system iterated over 100 000 records and verified their state. At this moment, we can see that the system requires more than six minutes per operation, while the Git system requires constant execution time because no checks are active, but the ES only updates the file in the Git repository. We can see how the storage demand linearly grows as the file gets updated. The measurements are shown in Table 5.7 and the graphs for the large data sets is shown in the Appendix A.

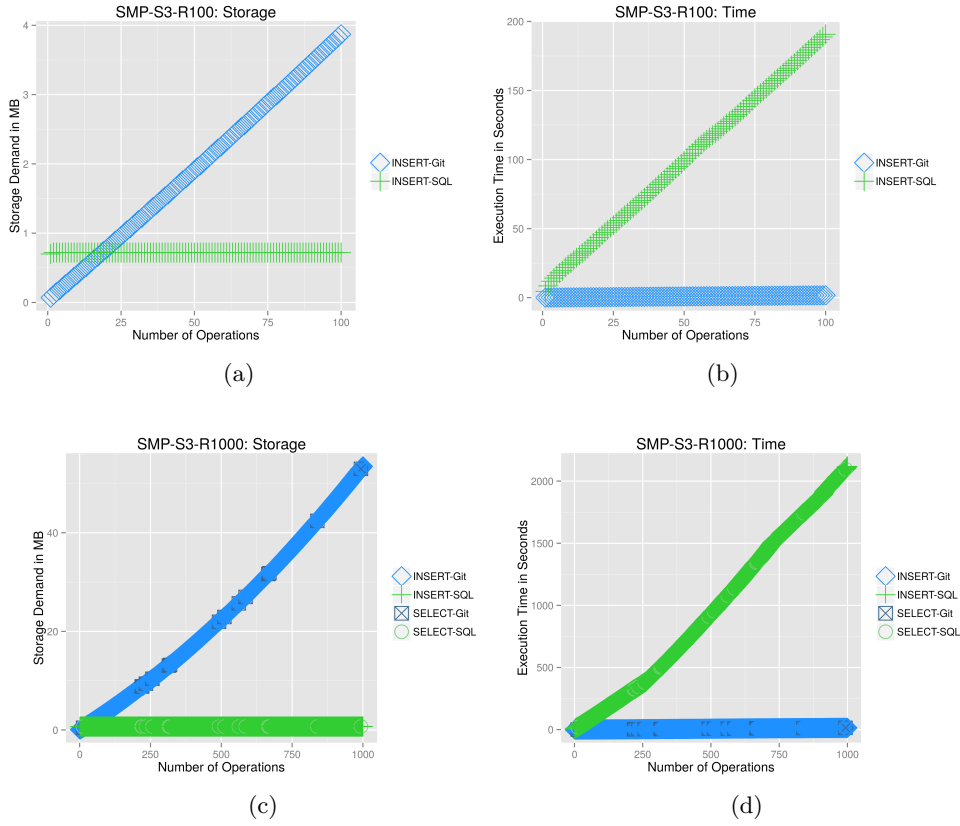


Figure 5.10: Evaluation SMP-S3

Complexity	Size PS (MB)	Size ES (MB)	Time PS (sec)	Time ES (sec)
Simple (SMP-100)	0.7	3.87	178.88	1.74
Simple (SMP-1000)	0.7	53.18	1485.27	17.73
Medium (MED-100)	30	420.55	1599.03	76.07
Medium (MED-1000)	30	2047.82	16679.23	769.52
Large (LRG-100)	416.0	8217.76	43509.92	1354.12

Table 5.7: S3 Evaluation Results

## 5.2.4 Evaluation SMP-S4

The graphs in the Figure 5.11 show the scenario of ephemeral data, where data is frequently inserted, changed and deleted. Again the tradeoff between the storage demand of the Git system and the execution time of the SQL system are displayed. We can see the same pattern for the simple case in Figure 5.11, for the average case in Figure A.8 and the for the complex case in Figure A.9. Both figures are available in the Appendix A. Table 5.8 shows the results.

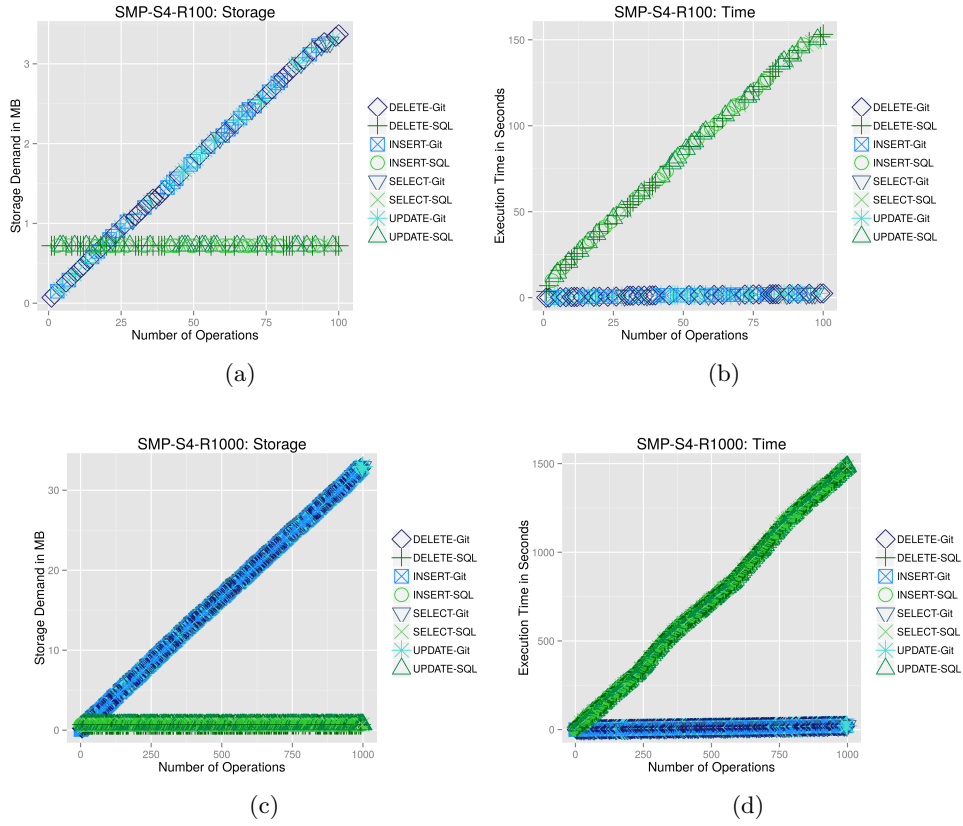


Figure 5.11: Evaluation SMP-S4

Complexity	Size PS (MB)	Size ES (MB)	Time PS (sec)	Time ES (sec)
Simple (SMP-100)	0.7	3.37	153.25	2.53
Simple (SMP-1000)	0.7	32.99	1485.27	23.68
Medium (MED-100)	30	389.75	1289.22	81.25
Medium (MED-1000)	30	2046.13	15442.67	790.55
Large (LRG-100)	416.0	7311.25	11497.82	1001.73

Table 5.8: S4 Evaluation Results

### 5.3 Suggestions and Potential Improvements

The aim of this evaluation was to measure the performance of the PS in comparison to the ES. In order to establish comparable conditions, we refrained from optimising scenario and application specific settings for both implementations. In the following

section, we give an overview how the performance can be improved for both systems, when specific features of both systems are used.

### 5.3.1 Reducing the Storage Demand for Git

On average, a CSV test file in scenario S1 occupies about 116 kB. Git stores each of these objects compressed for reducing storage space, but does not yet use delta compression. Listing 5.7 shows a directory listing of the Git repository for the *SMP-S3-R1000* scenario. After the execution, the repository contains data objects for each revision of the file. The 1000 operations on the CSV file contribute to a Git repository size of 65 MB.

Listing 5.7: Internal data structure of Git

```
1 EvalVM-1: /tmp $ du -h Evaluation_Git_Repo/
2 4,0K    Evaluation_Git_Repo/.git/branches
3 4,0K    Evaluation_Git_Repo/.git/hooks
4 176K    Evaluation_Git_Repo/.git/logs/refs/heads
5 180K    Evaluation_Git_Repo/.git/logs/refs
6 356K    Evaluation_Git_Repo/.git/logs
7 212K    Evaluation_Git_Repo/.git/objects/77
8 292K    Evaluation_Git_Repo/.git/objects/4c
9 288K    Evaluation_Git_Repo/.git/objects/71
10 ....
11 ....
12 ....
13 300K    Evaluation_Git_Repo/.git/objects/9d
14 64M    Evaluation_Git_Repo/.git/objects
15 4,0K    Evaluation_Git_Repo/.git/refs/tags
16 8,0K    Evaluation_Git_Repo/.git/refs/heads
17 16K    Evaluation_Git_Repo/.git/refs
18 65M    Evaluation_Git_Repo/.git
19 65M    Evaluation_Git_Repo/
```

Obviously, Git does not store only the deltas in this step, but rather creates copies of changed files during the operations. Git offers the manual execution of the garbage collection process, which iterates over the objects and creates more storage efficient deltas of the data object revisions. Listing 5.8 shows the manual execution, which creates so-called pack files. Pack files combine several objects and store them in a compressed binary format, which utilises the delta algorithm.

Listing 5.8: Internal data structure of Git

```
1 $ git gc --aggressive
2 Counting objects: 2970, Done.
3 Compressing objects: 100% (1980/1980), Done.
4 Writing objects: 100% (2970/2970), Done.
5 Total 2970 (delta 989), reused 0 (delta 0)
```

On the evaluation system, this operation needed 12 seconds. After this maintenance step, the repository size is reduced to only 928kB. The repository is locked during this operation. While this housekeeping task<sup>3</sup> improves storage management drastically, we did not implement the manual garbage collection in the prototype, as it is a manual maintenance task. The default value when the Git garbage collection is executed defaults to 6700 single objects<sup>4</sup>, which was beyond the limit of this evaluation. Adapting this value reduces the storage demand for the Git prototype ES.

<sup>3</sup><https://git-scm.com/docs/git-gc>

<sup>4</sup>Details: [https://git-memo.readthedocs.io/en/latest/garbage\\_collecting.html](https://git-memo.readthedocs.io/en/latest/garbage_collecting.html)

Run	Records	Storage Demand (MB)	Execution Time (sec)
Simple (SMP-S3-100) without indices	1000	0.7	178.88
Simple (SMP-S3-100) with indices	1000	8.0	480.34

Table 5.9: With and without index comparison

Query Indices	Execution Time (sec)	Storage Demand (MB)
None	1.196	60
Metadata columns	0.975	76
Queried columns	0.926	88
All columns	0.996	196

Table 5.10: Index storage demand

### 5.3.2 Improving the Query Execution Time for the MySQL Approach

For evaluating the query performance of the PS in comparison with the ES, we generated queries which cannot use database indices. For this reason the system was performing a full table scan for each operation which explains the slow query performance of the PS. As the data was growing in some of the scenarios and as more and more historical data was included into the MySQL tables, the execution performance was decreasing.

In a real world scenario, using indices is key to optimal performance. Database indices are a fundamental feature of relational database management systems. Instead of scanning the data row by row, MySQL utilises a B-tree index which speeds up the data retrieval. Deciding which columns should be indexed is scenario specific and requires thorough analysis of the queries which are executed.

Using indices is not free. While they boost performance for SELECT queries, they slow down INSERT, UPDATE and DELETE queries, because the system needs to update the index structures. Table 5.9 show this tradeoff for the SMP-S3-100 scenario, which mainly consists of index operations.

Adding indices to all columns in general has a negative effect, as insertion and deletion become more expensive. Table 5.10 shows results for a sample query without an index and with indices on a sample data set with 100 000 records. We can see the improving query performance for the metadata indices and when we explicitly add indices on those columns which are used for filtering.

## 5.4 Application Recommendations

As we have shown, both approaches offer advantages and disadvantages. The PS based on MySQL is the more sophisticated approach. It keeps the storage requirement low

and scales well for large data sets. When structure of the queries is known in advance, the performance of the system can be tuned by utilising column indices. As this approach natively utilises SQL as query language, also complex subsets can be created by formulating the appropriate queries. SQL is a standardised, very well known and widely used language. As the interface interacts directly with the database, subsets can also be built without SQL knowledge, which makes this approach feasible for a broad audience.

Because the data is stored in the MySQL database system already, it can easily be integrated into other workflows, which use MySQL as data storage backend. The table and database design is kept simple and can be implemented into existing MySQL instances, which makes this approach favourable for environments, where already a database infrastructure exists. For this reason we recommend the PS approach for scenarios which already utilise a database system and where the knowledge how to setup and administrate such systems is already available. On the downside, the PS does require a more complex setup than the ES. A database instance needs to be installed, maintained and administrated. This includes user management and user privileges, which may change during the life cycle of the system.

The ES was developed for testing the performance of the PS in comparison with a minimalistic system. The approach is based upon Git, which requires very little effort in installing and setup. This makes the ES an ideal approach for storing reproducible data sets in local or small scale research environments or even individual research data repositories. Git allows storing files not only locally, but also in remote repositories, which makes this approach likewise feasible for distributed teams. We integrated the ES tightly with the query store in order to be able to produce the same subsets. But the ES is not limited to the query store we implemented for the PS. The advantage of the ES is that it can be used with literally any query or scripting languages. This increases the flexibility of this approach and allows researchers not only to extract data with SQL but also with different systems. As long as these scripts used for creating subsets are preserved in a versioned fashion as well (for instance with Git) and the underlying software packages are available, these subsets can be made reproducible as well. As shown above, the ES produces relatively large repositories if not maintained properly.

## 5.5 Evaluation Summary

In this section we evaluated MySQL based PS implementation against a second system based on the source code versioning system Git, denoted ES. Both systems use SQL as the query language for creating the subset. This allows us to implement both systems with the same Query Store, which is responsible for preserving the subset creation metadata. We generated random CSV data of different complexity in terms of columns, average record length and amount of records and imported the data into both systems. In the following, we randomly executed SELECT, INSERT, UPDATE and DELETE statements on the CSV data and propagated the changes in both systems. We compared for both systems the storage demand and the execution time for the set of operations. After each

SELECT operation we exported the result as CSV file for each system and compared the files for identity by computing a hash value therefore.

Both systems are suitable providing reproducible subsets of CSV data. As shown in the evaluation graphs, there exists a tradeoff between the storage demand of the Git based ES and the time consumption of the PS. The reason for this tradeoff is that Git stores snapshots for each commit and this storage demand grows linearly with the amount of commits. The SQL system in contrast stores only the actual change in the form of one new additional record, which requires much less storage space. The MySQL database system uses the so called table space to store the data, which consists of pages of a defined size. This is the reason why the storage demand of the PS seems to be constant although new data is added or records are removed.

As explained in Section 5.1.9, there exist limitations for this evaluation. The PS offers much more features than the simple evaluation system we implemented for the purpose of the evaluation. For this reason the PS performs more tasks per operation than the ES, leading to a higher execution time demand in comparison to the ES.

The MySQL based implementation offers much room for improvement and can be optimised to perform faster for the different scenarios. In a real world application the format of the CSV files would usually be known in advance, allowing to use indices on specific columns which improve the performance.



# Conclusions and Outlook

## 6.1 Summary of the Thesis

So far, researchers have been lacking tools, which allow them to create subsets in a reproducible way, where peers can follow the creation process of subsets and understand how a researcher made a specific selection of the data. In many cases, researchers needed to rely on complex verbal descriptions of the subsetting process in the methods section of papers. It is a fundamental principle of the scientific method, that results are reproducible. So far hardly any experiment in the domain of computer science is reproducible [48]. Reproducibility needs to be achieved on several layers. In this work, we focused on the data which serves as input for experiments and therefore needs to be identifiable in a reliable way. In the same way researchers produce data sets as output, which must be identifiable for several reasons. In the first place, the output of one experiment may serve as input for other experiments down stream the processing line. Secondly researchers require to receive attribution for their work. This is only possible if the data they are using and the corresponding subsets are identifiable and citable.

In this thesis, we implemented the RDA guidelines for data citation with a prototype for making structured data citable. This is especially valuable for the long tail of research data, where complex systems are not available. The system we developed provides an easy to use Web interface, which allows researchers to upload existing CSV data files and create subsets from this data sources individually. Our prototype system automatically migrates the CSV files into a dynamically created relational database scheme and adds metadata on the fly. The data is stored in a versioned fashion, thus researchers can change (update), add (insert) or delete records into an existing data set without losing information about previous versions of their data set. Researchers can then use an interactive table for selecting subsets from their uploaded CSV data set, based on individual selection criteria, filters and specific sortings.

We make these steps traceable by persisting the operations a researcher applied during the creation of the subset. We store the parameters used and the filters and sortings

applied. This allows peers to understand how a researcher created a data set and what records have been included or excluded from a subset. This constitutes valuable metadata for a subset and improves the reproducibility of research experiments in general and data sets in particular.

The novelty in our approach is that the subsetting process is based on database queries. Instead of exporting and archiving each subset individually for later access, our system allows re-executing queries against versioned data and thus retrieve historical data sets on demand. This reduces the required storage size for subsets, as individual data exports do not have to be stored, archived, maintained and kept accessible. The query based approach is a lightweight method for providing subsets of data in a dynamic way. As we store the data in a version fashion, even subsets from evolving and frequently changing data sets can be made reproducible.

The prototype we developed is the first implementation of the RDA Recommendations for Data Citation. We described the recommendations in detail and mapped each recommendation to a specific feature of our implementation. By evaluating the system we can provide feedback for the further development of the recommendations.

The evaluation of the prototype has shown that storing and handling CSV files as a significant advantage in terms of storage demand. Storing the data in a versioned database produces very little overhead compared to storing individual subsets as files or even in comparison with Git.

## 6.2 Future Work

This thesis provides the first implementation of the RDA Recommendations for Dynamic Data Citation. As the recommendations are in an evolving state themselves, improvements of this work are a logical consequence. The focus of this work was a prototype for structured data, our aim was to keep the implementation simple and avoid complexity wherever possible.

There are several ways how to improve the prototype in the future. Currently the interface only allows querying for textual matches in each columns by using simple text comparison operations. Although we support wild cards, the system currently does not implement data type specific queries and range queries. At the moment we also support only one filter per column. Allowing users to utilise several filters for each column would allow users to specify more complex subsets based on highly specific subset criteria.

Data versioning is also done in the simplest form possible. We store each CSV file as a dedicated table in a predefined database schema. Record versioning is done by adding timestamps for each event which triggered the creation of a new version. We store the versions of a record in the same table as the most actual value. Moving the historical data in separate history tables improves the data access on most recent versions of the records, which is the common use case in most scenarios, where users are not interested in a specific, historical version. This could be considered in future releases.

The user interface utilises an Ajax-based interactive querying mode, which retrieves the most recent revision of a record in the data set. As we store all versions of each record,

querying the data becomes slower with an increasing number of historical records. A more complex database scheme allowing to utilise database specific querying techniques would reduce the time needed to retrieve the most recent versions and improve the usability for users.



# Evaluation Graphs

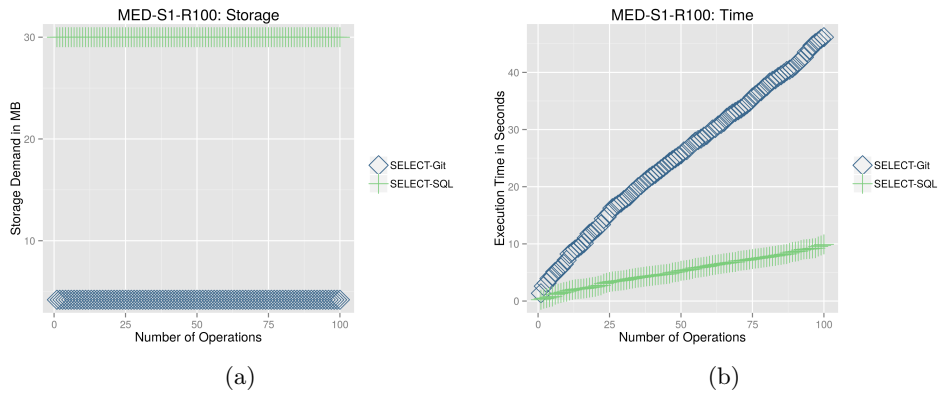


Figure A.1: Evaluation MED-S1-100

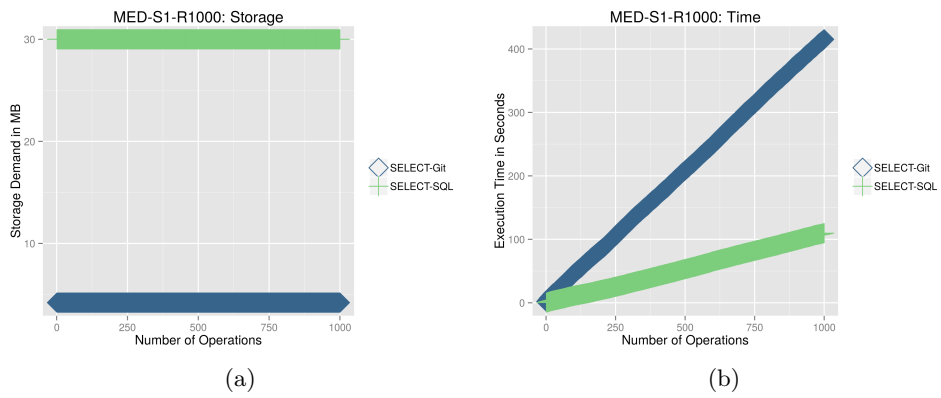


Figure A.2: Evaluation MED-S1-1000

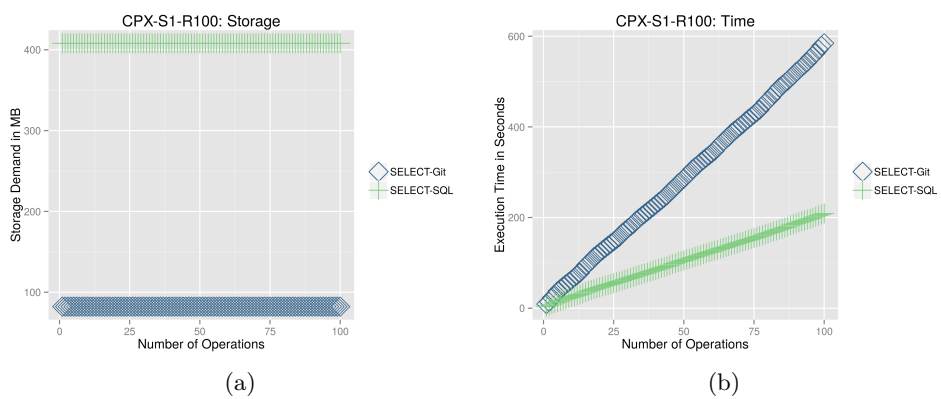


Figure A.3: Evaluation LRG-S1

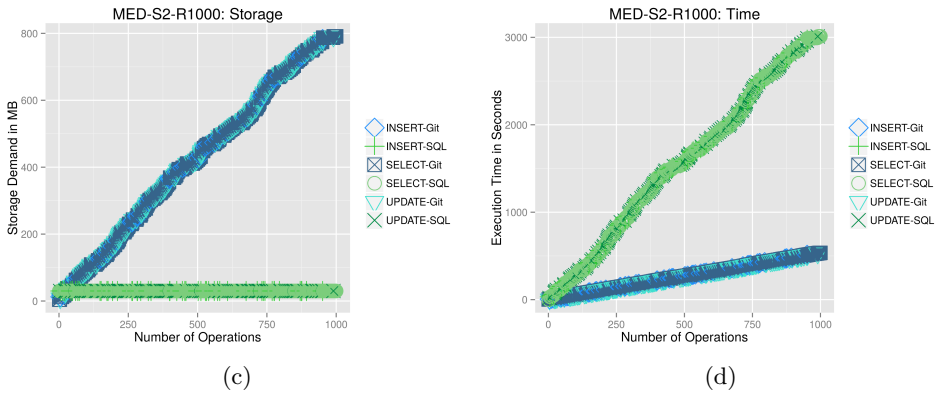
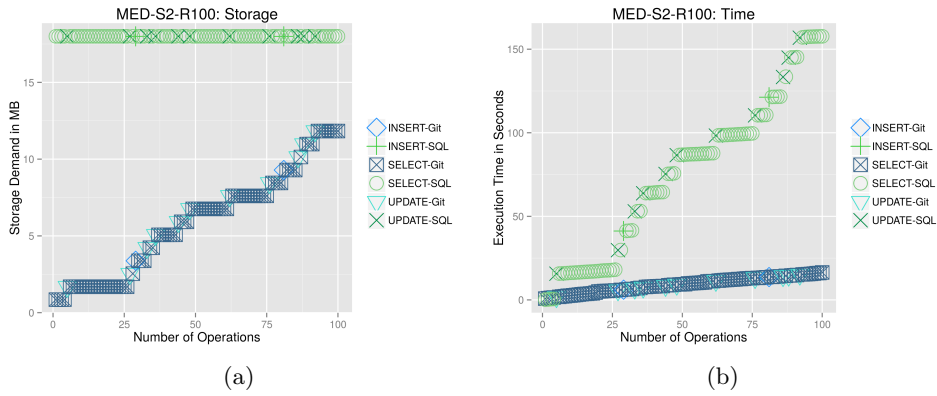


Figure A.4: Evaluation MED-S2

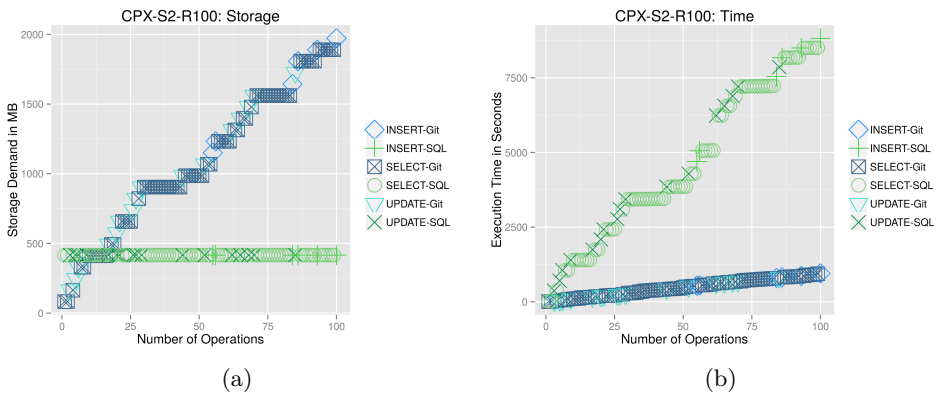


Figure A.5: Evaluation LRG-S2

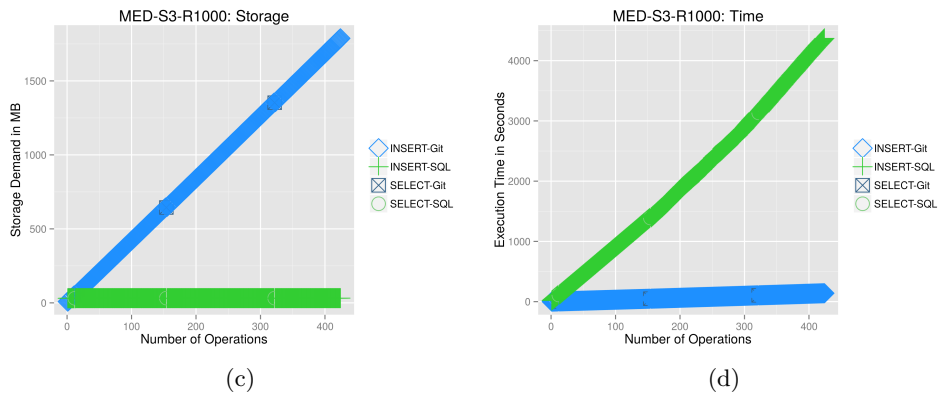
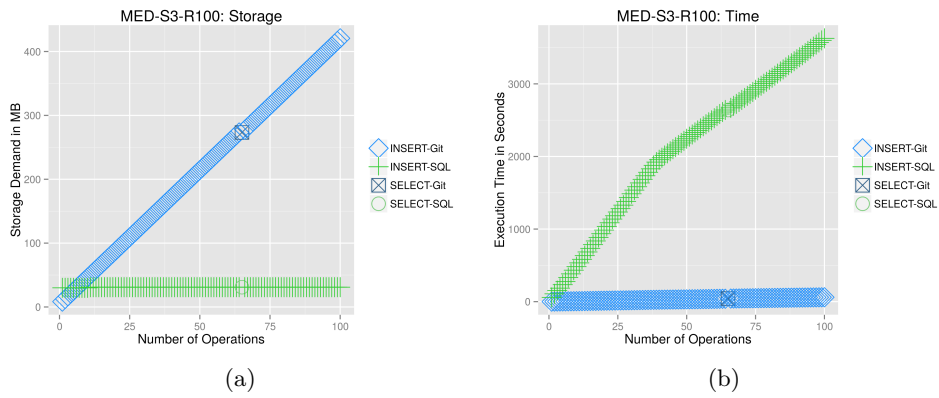


Figure A.6: Evaluation MED-S3

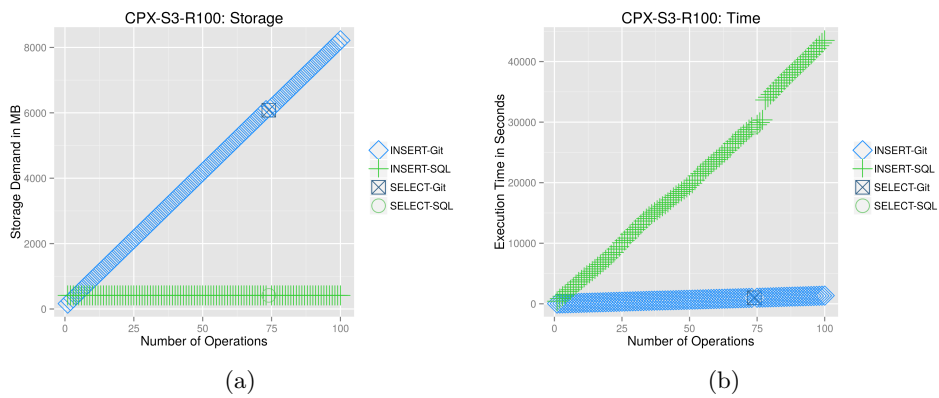


Figure A.7: Evaluation LRG-S3



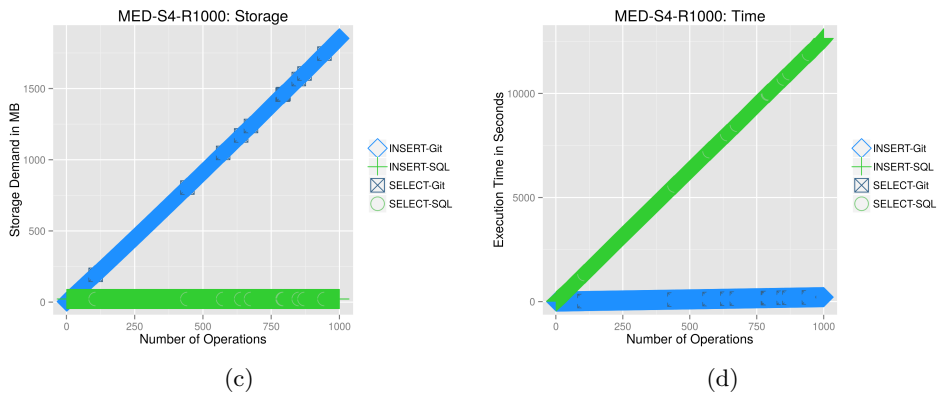
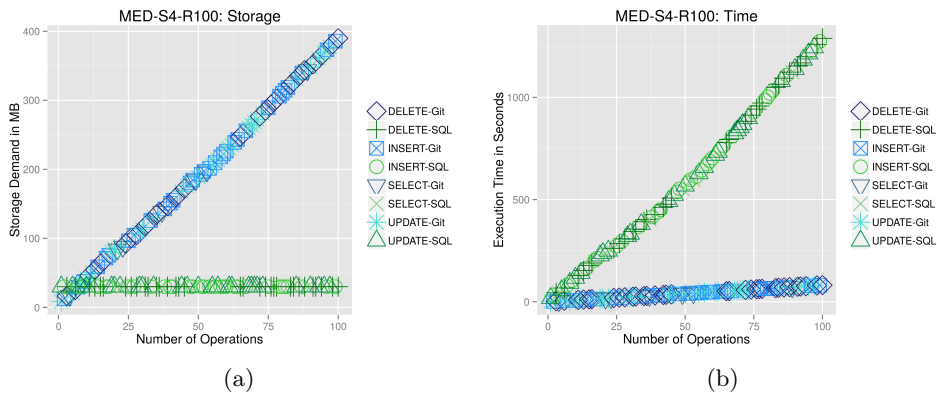


Figure A.8: Evaluation MED-S4

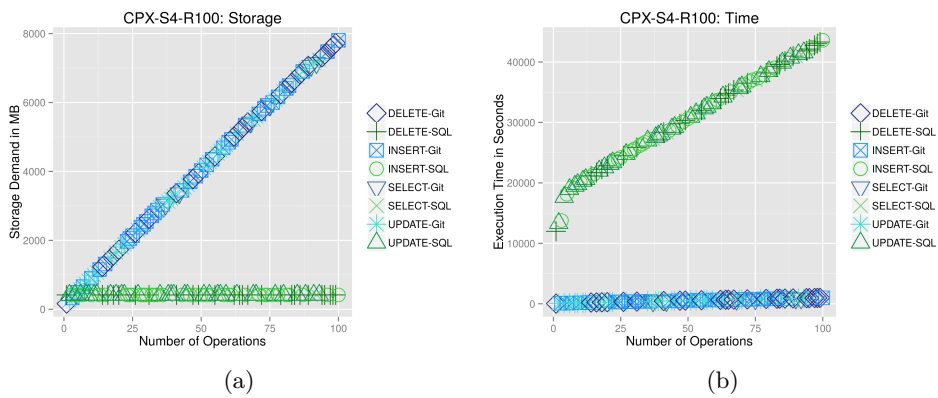


Figure A.9: Evaluation LRG-S4



# Bibliography

- [1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
- [2] J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.
- [3] A. A. Alsheikh-Ali, W. Qureshi, M. H. Al-Mallah, and J. P. Ioannidis. Public availability of published research data in high-impact journals. *PloS one*, 6(9), 2011.
- [4] M. Altman and M. Crosas. The evolution of data citation: From principles to implementation. *IAssist Quarterly*, 63, 2013.
- [5] Australian National Data Service. Data Citation Awareness, Aug. 2011. Available online: <http://ands.org.au/guides/data-citation-awareness.pdf>.
- [6] E. Bellini, C. Cirinna, and M. Lunghi. Persistent identifiers distributed system for cultural heritage digital objects. In *Fifth International Conference on Digital Preservation (iPRES 2008)*, September 2008.
- [7] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.
- [8] D. Bollier and C. M. Firestone. *The promise and peril of big data*. Aspen Institute, Communications and Society Program Washington, DC, 2010.
- [9] J. Brase. DataCite - A Global Registration Agency for Research Data. In *COINFO 2009: Proceedings of the Fourth International Conference on Cooperation and Promotion of Information Resources in Science and Technology*, Washington, DC, USA, 2009. IEEE Computer Society.
- [10] P. Buneman, S. Davidson, and J. Frew. Why data citation is a computational problem. *Communications of the ACM (CACM)*, 2016. Preprint available at: <http://frew.eri.ucsb.edu/private/preprints/bdf-cacm-data-citation.pdf>.
- [11] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan. Archiving scientific data. *ACM Transactions on Database Systems (TODS)*, Volume 29(1):2–42, March 2004.

- [12] B. Byrne and P. Triyambakaaradhya. Temporal support in relational databases. In *10th International Workshop on the Teaching, Learning and Assessment of Databases*. The Higher Education Academy, 2012.
- [13] S. Callaghan, S. Donegan, S. Pepler, M. Thorley, N. Cunningham, P. Kirsch, L. Ault, P. Bell, R. Bowie, A. Leadbetter, et al. Making data a first class scientific output: Data citation and publication by NERC’s environmental data centres. *International Journal of Digital Curation*, 7(1), 2012.
- [14] CCSDS Secretariat. Reference Model for an Open Archival Information System (OAIS). Technical report, CCSDS, June 2012. 650.0-M-2.
- [15] R. Chatterjee, G. Arun, S. Agarwal, B. Speckhard, and R. Vasudevan. Using data versioning in database application development. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 2004., pages 315–325, May 2004.
- [16] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
- [17] E. F. Codd. Relational database: a practical foundation for productivity. *Communications of the ACM*, 25(2):109–117, 1982.
- [18] P. Dadam, V. Lum, and H. Werner. Integration of time versions into a relational database system. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pages 509–522, 1984.
- [19] C. J. Date and H. Darwen. The SQL standard. *SQL/92 mit den Erweiterungen CLI und PSM*, 1993.
- [20] R. C. Davis. Git and github for librarians. *Behavioral and Social Sciences Librarian*, 34(3):158–164, 2015.
- [21] Y. Demchenko, P. Grosso, C. De Laat, and P. Membrey. Addressing big data issues in scientific data infrastructure. In *International Conference on Collaboration Technologies and Systems (CTS)*, pages 48–55, May 2013.
- [22] C. Dyreson, F. Grandi, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, et al. A consensus glossary of temporal database concepts. *ACM Sigmod Record*, 23(1):52–64, 1994.
- [23] A. Goodman, A. Pepe, A. W. Blocker, C. L. Borgman, K. Cranmer, M. Crosas, R. Di Stefano, Y. Gil, P. Groth, M. Hedstrom, et al. Ten simple rules for the care and feeding of scientific data. *PLoS Computational Biology*, 10(4):e1003542, 2014.
- [24] P. Gulutzan and T. Pelzer. *SQL-99 complete, really*. CMP books, 1999.
- [25] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

- [26] J. K. Hans-Werner Hilse. *Implementing Persistent Identifiers: Overview of concepts, guidelines and recommendations*. Consortium of European Research Libraries, European Commission on Preservation and Access, 2006.
- [27] M. Hedstrom. Digital preservation: a time bomb for digital libraries. *Computers and the Humanities*, 31(3):189–202, 1997.
- [28] A. J. Hey and A. E. Trefethen. *The data deluge: An e-science perspective*. Wiley and Sons, 2003.
- [29] T. Hey, S. Tansley, K. M. Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*, volume 1. Microsoft Research Redmond, WA, 2009.
- [30] W. C.-Y. Hsieh, A. Lloyd, and E. H. Veach. Systems and methods of increasing database access concurrency using granular timestamps, June 2013. US Patent App. 13/909,928.
- [31] C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass. A glossary of temporal database concepts. *ACM Sigmod Record*, 21(3):35–43, 1992.
- [32] C. S. Jensen and R. T. Snodgrass. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, 1999.
- [33] G. Klyne and C. Newman. Date and Time on the Internet: Timestamps. RFC 3339 (Proposed Standard), available at <http://www.ietf.org/rfc/rfc3339.txt>, July 2002.
- [34] J. Kratz and C. Strasser. Data publication consensus and controversies. *F1000Research*, 3, 2014.
- [35] T. S. Kuhn. *The structure of scientific revolutions*. University of Chicago press, 2012.
- [36] K. Kulkarni and J.-E. Michels. Temporal features in SQL: 2011. *ACM Sigmod Record*, 41(3):34–43, 2012.
- [37] J. Kunze. Towards electronic persistence using ARK identifiers. In *Proceedings of the 3rd ECDL Workshop on Web Archives*, 2003.
- [38] J. Kunze. The ARK identifier scheme. RFC Draft Version 18, available at <http://tools.ietf.org/html/draft-kunze-ark-18>, Apr 2013.
- [39] P. B. Ladkin. Primitives and units for time specification. In *Proceedings of the Fifth Association of the Advancement of Artificial Intelligence (AAAI) National Conference on Artificial Intelligence*, pages 353–359, 1986.
- [40] S. Lawrence, L. C. Giles, and K. Bollacker. Digital Libraries and Autonomous Citation indexing. *Computer*, 32(6):67–71, 1999.

- [41] D. M. Levy. Where’s Waldo? Reflections on copies and authenticity in a digital environment. *Authenticity in a Digital Environment*, pages 24–31, 2000.
- [42] S. Lyons. Persistent identification of electronic documents and the future of footnotes. *Law Library Journal*, 97, 2005.
- [43] B. C. Martinson, M. S. Anderson, and R. De Vries. Scientists behaving badly. *Nature*, 435(7043):737–738, 2005.
- [44] S. M. Mojon-Azzi and D. S. Mojon. Scientific misconduct: from salami slicing to data fabrication. *Ophthalmologica*, 218(1):1–3, 2004.
- [45] H. Mooney and M. P. Newton. The anatomy of a data citation: discovery, reuse, and credit. *Journal of Librarianship and Scholarly Communication*, 1(1):6, 2012.
- [46] C.-I. T. G. 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. *Data Science Journal*, 12:CIDCR1–CIDCR75, 2013.
- [47] C. H. Papadimitriou and M. Yannakakis. On the complexity of database queries. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 12–19. ACM, 1997.
- [48] R. D. Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011.
- [49] S. Proell, K. Meixner, and A. Rauber. Precise data identification services for long tail research data. In *13th International Conference on Digital Preservation (iPRES 2016)*, 10 2016.
- [50] S. Proell and A. Rauber. Citable by Design - A Model for Making Data in Dynamic Environments Citable. In *2nd International Conference on Data Management Technologies and Applications (DATA2013)*, Reykjavik, Iceland, July 29-31 2013.
- [51] S. Proell and A. Rauber. Data Citation in Dynamic, Large Databases: Model and Reference Implementation. In *IEEE International Conference on Big Data 2013 (IEEE BigData 2013)*, Santa Clara, CA, USA, October 2013.
- [52] S. Proell and A. Rauber. A Scalable Framework for Dynamic Data Citation of Arbitrary Structured Data. In *3rd International Conference on Data Management Technologies and Applications (DATA2014)*, Vienna, Austria, August 29-31 2014.
- [53] N. Provos and D. Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.
- [54] K. Ram. Git can facilitate greater reproducibility and increased transparency in science. *Source code for biology and medicine*, 8(1):7, 2013.

- [55] A. Rauber, A. Asmi, D. van Uytvanck, and S. Proell. Data Citation of Evolving Data - Recommendations of the Working Group on Data Citation. <https://rd-alliance.org/group/data-citation-wg/outcomes/data-citation-recommendation.html>, October 2015.
- [56] A. Rauber, A. Asmi, D. van Uytvanck, and S. Proell. Identification of reproducible subsets for data citation, sharing and re-use. *Bulletin of IEEE Technical Committee on Digital Libraries, Special Issue on Data Citation*, May 2016.
- [57] Y. Shafranovich. Common format and mime type for comma-separated values CSV files. RFC 4180 (Informational), October 2005. Updated by RFC 7111.
- [58] J. E. Sieber and B. E. Trumbo. (not) giving credit where credit is due: Citation of data sets. *Science and Engineering Ethics*, 1(1):11–20, 1995.
- [59] R. Snodgrass and I. Ahn. A taxonomy of time databases. *ACM Sigmod Record*, 14(4):236–246, 1985.
- [60] S. Sun, L. Lannom, and B. Boesch. Handle system overview. Technical report, 2003.
- [61] E. Tonkin. Persistent identifiers: considering the options. *Ariadne*, (56):8, 2008.
- [62] K. Torp, C. S. Jensen, and R. T. Snodgrass. Effective timestamping in databases. *The VLDB Journal*, 8(3-4):267–288, February 2000.