Precise Data Identification Services for Long Tail Research Data

Stefan Pröll
SBA Research
Vienna, Austria
sproell@sba-research.org

Kristof Meixner
Vienna University of Technology
Vienna, Austria
kristof.meixner@tuwien.ac.at

Andreas Rauber
Vienna University of Technology
Vienna, Austria
rauber@ifs.tuwien.ac.at

ABSTRACT

While sophisticated research infrastructures assist scientists in managing massive volumes of data, the so-called long tail of research data frequently suffers from a lack of such services. This is mostly due to the complexity caused by the variety of data to be managed and a lack of easily standardiseable procedures in highly diverse research settings. Yet, as even domains in this long tail of research data are increasingly data-driven, scientists need efficient means to precisely communicate, which version and subset of data was used in a particular study to enable reproducibility and comparability of result and foster data re-use.

This paper presents three implementations of systems supporting such data identification services for comma separated value (CSV) files, a dominant format for data exchange in these settings. The implementations are based on the recommendations of the Working Group on Dynamic Data Citation of the Research Data Alliance (RDA). They provide implicit change tracking of all data modifications, while precise subsets are identified via the respective subsetting process. These enhances reproducibility of experiments and allows efficient sharing of specific subsets of data even in highly dynamic data settings.

Keywords

Data Identification, Data Citation, Reproducibility, Long Tail Research Data

1. INTRODUCTION

Human beings in general and researchers in particular are said to be lazy when tedious tasks not directly related to the primary research endeavour are due. Unfortunately, this includes providing metadata for data sets, storing, archiving and citing the data used in a research paper. Without being able to share this data, we are creating data silos, which hinder repeatability and verifyability of experiments and the re-use of data. In this work, we present three methods for improving the identification of subsets of dynamic data, by automating obnoxious tasks. Our goal is to address data identification in small and big data scenarios. Specifically, we focus on comma separated value (CSV) files, which are prevalent in both settings. Recent open data initiatives such as in the UK[1], USA[2] or Austria[3] 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. More than 50 % of the data sets from the open data portals of the UK and Austria for instance are available in CSV[4]. The CSV format is used for exchanging data and often provided as data export from more complex database systems. Despite the relatively small size of individual CSV files, handling massive numbers of CSV files in multiple versions is a challenge in big data scenarios [1].

1.1 Little Science, Big Science

Contrary to many high-volume big data settings, where standardised infrastructure is available, there exist other big data settings with less mature processes, due to the lack of tools, resources and community exchange. This area is denoted the long tail of research data and subsumes large portions of data that are highly heterogeneous, managed predominantly locally within each researcher's environment, and frequently not properly transferred to and managed within well-curated repositories. The reason is that in the so called little science [2, 3], common standards and defined best practices are rare. This is particular true in research disciplines, which do not yet have a tradition of working with advanced research infrastructures and many data sets still reside on the machine of the researcher. Being able to identify which data set served as the input for a particular experiment, is based on the rigour of the scientists, and their ability to identify the particular data set again, often without proper tool support.

Reproducibility is a core requirement in many research settings. It can be tackled from several perspectives, including organisational, social and technical views. For researchers, the authors of [4] introduced 10 rules for making computational results reproducible, by describing all intermediate steps and storing the data artifacts used as inputs and produced as outputs. Both worlds - small and large scale data experiments - share the difficulty of precisely identifying data sets used as input and produced as output. This can be attributed to two main reasons: the dynamics frequently encountered in evolving data sets and tendency of scientists for using specific subsets for specific analysis that need to be precisely identified. Whereas the identification of a data set in smaller scale settings can be figuratively compared to the search of a needle in the hay stack, identifying evolving data sets in large scale environments is rather the search for the needle in a silage.

[1] http://data.gov.uk

[2] http://www.data.gov

[3] https://www.data.gv.at

[4] Data collected from the portals at 22.04.2016

1.2 Versioning Research Data

As new data is being added and corrections are made in existing data sets, we face the questions of how intermediate revisions of a data source can be efficiently managed. Having this data available, i.e. being able to obtain an earlier version of a data set, is a fundamental requirement for the reproducibility of research processes. Access to earlier versions is also essential to support comparability of experiments by running different experiments on identical data. Thus, maintaining and accessing dynamically changing research data is a common challenge in many disciplines. Storing duplicates of data sets, the prevalent approach to address this problem, hardly scales with large and distributed data volumes, while increasing the complexity of having to manage enormous amounts of data files. This calls for more efficient ways of handling versions of evolving data files.

1.3 Creating Subsets

Creating a subset is based on implicit information which records to include into a data set and which ones to omit. The basic methods needed for creating subsets are filtering and sorting. So far this process is hardly captured and researchers tend to store subsets as individual files, causing high redundancy problems and leading to an explosion of individual data files to be managed. Alternatively, researchers may chose to refer to the entire data set and provide a natural language description of which subset they were using. Albeit, this description is frequently ambiguous and may require the reader to invest significant effort to recreate the extract same subset, while making it very hard to verify whether the resulting subset is identical.

In this work we present an approach allowing to efficiently identify data sets and derived subsets, even if the data source is still evolving. These identification services can be integrated into scientific workflows, and therefore allow to unambiguously pinpoint the specific subset and version. Our approach is based upon versioning and timestamping the data as well as query based subsets of data being used in an experiment or visualisation. In our approach, we interpret *query* in a rather broad way, as by query, we understand any descriptive request for a subset of data. A query can either be an actual database query, or any operation allowing to retrieve a subset from a data source using, for example, scripts. Instead of creating duplicates of data, we use queries for (re-) constructing subsets on demand. We trace the subsetting process and assign persistent identifiers (PID) to these queries instead of static data sets. With this mechanism, we provide reproducible data sets by linking the PID to the subset creation process and matching the data against a versioned state of the source data set. This approach has been released as a recommendation by the RDA Working Group on data citation [5] and refined in [6] to address efficient and precise identification and citation of subsets of potentially highly dynamic data.

We present three implementations of this approach supporting CSV data and compare their respective advantages and disadvantages. The first approach is based on a simple file-system based versioning with script-able queries. The second approach is an extension of the first approach and based on Git branching, which enables users to work simultaneously with data sets without distracting each other. The third approach uses transparent migration of the CSV data into a relational database system, allowing more efficient versioning and more flexible query-based subset generation.

The remainder of this paper is organised as follows. Section 2 provides an overview of the state of the art from the areas of research data management, data citation and persistent identification. Section 3 outlines the challenges of dynamic data citation in research areas working with the long tale of research data. In Section 4 we introduce three realisations of the dynamic data citation method optimised particularly for small and medium-sized data sets distributed as CSV files. Section 5 provides the evaluation of the approaches, Section 6 provides the conclusions.

2. RELATED WORK

Citing publications has a century old tradition and its methods have been applied to modern scholarly communication including data sets [7, 8]. We need to be able to identify such data sets precisely. As URLs are not a long term option, the concept of persistent identifiers was introduced. Persistence is achieved by using centrally managed PID systems [9], which utilise redirection to resolve new locations of data files correctly. In many cases landing pages are the target of resolvers [10]. Landing pages contain metadata for human readers, but no standard solution regarding versioning and subsetting of data sets is provided, that is accessible by humans and machines. Recent developments try to enrich the mere redirection purpose identifier infrastructures by adding machine readable metadata [11] and providing the context of data sets [12]. We thus need to ensure that our solution can support these mechanisms of persistent data identification and citation by allowing the assignment of PIDs to data.

Current citation practices usually refer to static data files. However, we increasingly find situations where such data files are dynamic, i.e. new data may be added at certain intervals of time. For working with the data as it existed at a specific point in time (e.g. to verify the repeatability of an experiment, or to compare the result of a new method with earlier published results), we need to ensure that we can provide exactly the same data as input. To achieve this, data needs to be versioned, which is a common task in the data management domain [13] and implemented in software applications dealing with critical data [14]. With decreasing storage costs preserving previous versions even of high volume data has become a service offered by many data providers but still storing multiple versions is a challenge [15]. Storing previous versions of data sets is usually accompanied by timestamps [16]. Each operation which changes the data is recorded and annotated with a timestamp of its occurrence.

As mentioned above, natural language description frequently is not precise enough to unambiguously describe a specific subset. Storing redundant copies, on the other hand, does not scale well. Thus, the concept of a dynamic identification of subsets using query stores has been introduced [17]. The query store does not only store the queries as text, but also preserves the parameters of each query. This allows providing this information on other representations than the original query and enables to migrate the query to other systems. The query store operates on versioned data and queries [18], which allows retrieving only those versions of the records which have been valid during the original execution time. The data and the queries are versioned, the system can be used for retrieving subsets of large data sources exactly the same way as they have been at any given point in time [19].

The Research Data Alliance (RDA) Data Citation Working Group published 14 recommendations on how to make research data citable [6]. The RDA data citation mechanism can be used for evolving and for static data and is based upon versioned data and query mechanisms, which allow to retrieve previous versions of specific data sets again.

3. CHALLENGES IN HANDLING SMALLER SCALE RESEARCH DATA

Research disciplines and data in the so-called "long tail" are often suffering from a lack of professional tools and infrastructure which could support researchers in creating, using, sharing and verifying research data sets. If peer researchers want to repeat the process again or reuse the data to compare results of different approaches on the same data, means for verifying if the correct data were used are essential. Yet, this is far from trivial, with complexity caused primarily by two issues: dynamics in the data and the usage (and thus precise identification) of specific subsets of data.

3.1 Versioning Approaches: How Change is Traced

While researchers used to share static data files in the past, in current research settings the data we use is increasingly dynamic: new data being added, errors being corrected, wrong items being deleted. Ways this is dealt with include batch release of subsequent versions of the data, resulting in delayed release of corrections as they need to be aggregated until the next monthly, quarterly or annual release is due, as well as managing many redundant files, leading to high complexity in file naming and versioning conventions. Typically researchers utilise a rename and copy approach, where each version of a data set is distinguished by its file name. Recommendations for naming files exist [20], suggesting to use project or experiment name or acronym, coordinates, names, dates, version numbers and file extension for application-specific files. Nevertheless it is cumbersome and error prone for researchers. We thus need automated procedures allowing researchers to manage different versions of evolving data, allowing them to go back to earlier versions of data when needed. This should happen in an automated way, not putting the burden of version management and identification of changes on the researcher.

3.2 Creating Subsets From Implicit Information

Researchers often work with subsets from larger data sources, for curating specific aspects of a data set or visualising a specific view. Many publications only cite the full, raw data source and describe used subsets only superficially or ambiguously, by using natural language description for instance. From a reproducibility perspective, it is essential to know precisely, which subsets of data was used during a processing step. In contrast to large scale systems, which often guide researchers through standardised workflows of data filtering, the procedures in smaller scale research are often less well structured and defined. For this reason there is a larger variance in the way how subsets of data can be obtained and how subsets have been created. In larger scale settings, sophisticated database management systems are in place. In the small scale domain, text processing or spreadsheet programs are often used for creating a subset from a file. Scripting languages allow filtering, sorting and selecting subsets from file in a more automated way, but obtaining a specific subset again from a versioned data file in a reproducible way is a challenge.

For making implicit sub-setting information explicit, we need to trace the subset creation process itself and store this information in a persistent way. As manual work is susceptible to errors, an automated solution is a basic requirement for the integration of identification as a service into existing scientific workflows.

4. PRESERVING THE INFORMATION OF THE SUBSET CREATION PROCESSES

For this reason we introduce three implementations for the automated, unique identification of data, based on the data citation concepts introduced by [17, 19] and on the RDA recommendations for data citation [6]. This dynamic data citation is based upon two generic principles: (1) In order to be able to retrieve earlier versions of data, the underlying data source must be timestamped and versioned, i.e. any addition to the data is marked with a timestamp, and delete or update of a value is marked as delete and re-insert with respective timestamps. (2) Subsets are identified via a timestamped query, i.e. the query that was executed to generate a subset is retained in a so-called query store to enable its re-execution with the according timestamp. By assigning persistent identifiers to this information, understanding and retrieving the subset at a later point in time is possible. Integrating a query based identification service improves the reproducibility of scientific workflows in small and large scale scientific research likewise.

The three implementations of these principles for CSV files presented below differ primarily in their way of storing the data in the back end. Two approaches are based on Git, a wide spread version control system for source code, one approach utilises a migration process into a database system. The first approach uses a simple versioning scheme (Git) leading to low system complexity, but also less flexibility in subset generation and lower scalability. The second approach is also based on Git and utilises the branching model allowing simultaneous editing of data sets. The third approach migrates the CSV file transparently into a relational database, leading to higher complexity in system maintenance but providing higher efficiency and flexibility. In all three cases, the subset is identified via the query mechanisms (i.e. database queries via an API or graphical interface, scripting languages or via scrip-table SQL statements operating on the CSV file). The queries used in all three approaches are timestamped and associated with a PID. It is worth noting that we utilise a simplified PID approach in this paper, but the principle is compatible with accepted solutions such as DOI or other PID systems.

4.1 Using Git for Creating Reproducible Subsets

Source code management software and distributed revision control systems such as Git[5] or Subversion[6] are spreading from the software development departments to the labs,

[5] https://git-scm.com/

[6] http://subversion.apache.org/

as version control systems allow 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 each change of a file is committed into the repository, the changes are traceable and previous versions of each can be compared with the current revision.

Many different tools exist for manipulating CSV data, ranging from command line applications such as awk, sed, csvkit[7] to scriptable statistical software such as R.

In the following example use case, users provide a list of the Top500[8] super computers in a CSV file as input for the script. The list is updated periodically and each change is committed to the Git repository. A user interested in analysing the top-50 computers for some study creates an according subset and selects the columns Rank, Site and Cores from the file. The subset will be stored in the location provided as the second parameter to the script. Listing 1 provides a simple example for creating such a subset of CSV data using the mathematical software R. Listing 2 shows the execution of the script in a Linux shell.

Listing 1: Rscript for Subsetting

```
# Create a subset of the
# top 5 of the Top500 list
args <- commandArgs(trailingOnly = TRUE)
inputDatasetPath=args[1]
outputSubset=args[2]
dataset <- read.csv(inputDatasetPath,
    header=TRUE)
subset <- subset(dataset,
    Rank<=5,select=c(Rank,Site,Cores))
write.csv(subset, file=outputSubset)
```

Listing 2: Executing the Script

```
# Execute the R script and
# obtain a subset from the provided CSV file
/usr/bin/Rscript top5-subset.r \
/media/Data/Git-repository/ \
    supercomputing/supercomputer.csv \
/media/Data/Git-repository/ \
    supercomputing/supercomputer-top5.csv
```

We store these scripts in Git to retrieve the very same data set again, by executing the proper version of a script against the correct version of the data set. To do so, we store the CSV file name and location and the execution timestamp in a metadata/landing page file in the Git based query store. Each query is assigned a PID, which serves as file name of the according metadata file in the query store, which allows retrieving the data later by resolving the PID to the file name.

We implemented a prototype based on the Eclipse JGit Java library[9], which offers a low level API for the interaction with Git repositories. Revisions of the data set are committed to the repository, where Git stores a commit hash and the timestamp of the update. If users want to retrieve a subset again at a later point in time, they first retrieve the metadata file from the Git system using the PID as the file name. This file then provides the file name of the CSV data set and the execution timestamp of the query. In the next step, the system traverses the revision tree with the RevWalk object and builds a revision graph based on the commit dates[10]. We filter the commits and select the closest timestamp valid before the execution of the script[11]. This revision was valid during the execution of the original query. We fetch this version from the repository and re-execute the R script against the versioned data set, as depicted in Figure 1.

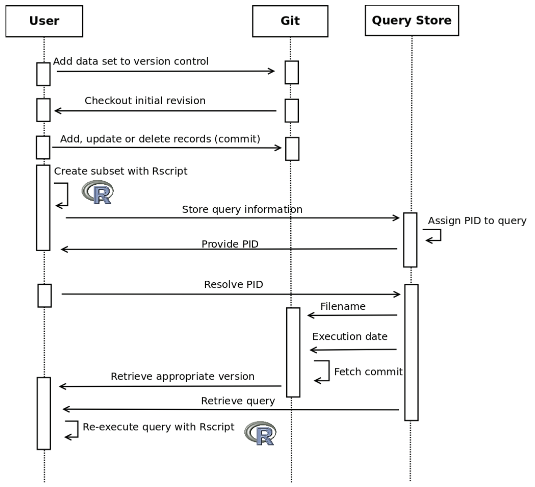

User
Git
Query Store
Add data set to version control
Checkout initial revision
Add, update or delete records (commit)
Create subset with Rscript
Store query information
Assign PID to query
Provide PID
Resolve PID
Filename
Execution date
Fetch commit
Retrieve appropriate version
Retrieve query
Re-execute query with Rscript

Figure 1: The CSV Subsetting Workflow with Git

For making this process reproducible, the user commits both, the CSV data file and the R script into the Git repository. The metadata files are committed into the Git repository in a separate PID folder. This folder contains all PID identified metadata files of reproducible data sets, using the PID as the file name. This allows us establishing a unique link between the PID and the metadata file, and by the transitivity, also with the data and the scripts. The metadata file contains the execution time, application version, the script and its parameters used as well as the re-execution steps for each subset. The metadata required can be generated automatically by using Git tools, no additional software dependencies are required. Listing 3 shows an example for the collected metadata and the references to versioned data and script files.

Listing 3: The Metadata File

```
# PID=1234/abcdefgh
# Repository_Path=/media/Data/Git-Repository
# Execution_Time=2015-09-30:11:07:09
# Subset_Tool=R scripting front-end version 3.2.2
    (2015-08-14)
# Subset_Tool_Path=/usr/bin/Rscript
# Input_Script_Path=supercomputing/top5-script.r
# Input_Script_Hash=bef5d...d7861:supercomputing/top5-
    script.r
# Dataset_Path=supercomputing/supercomputer.csv
# Dataset_Commit_Hash=acaed...4cf9c:supercomputer.csv
# Output_Path=/tmp/supercomputer-top5.csv

# Original execution:
#   /usr/bin/Rscript supercomputing/top5-script.r \
#   /media/Data/Git-repository/supercomputing/
    supercomputer.csv \
#   /tmp/supercomputer-top5.csv

# Recommended re-execution
# Retrieve script
git --git-dir=/media/Data/Git-Repository/.git/ \
show bef5d...d7861:supercomputing/top5-script.r \
> /tmp/reproduced-datasets/top5-script.r
# Retrieve data set
git --git-dir=/media/Data/Git-Repository/.git/ \
show 47bed...b9792:supercomputing/supercomputer.csv \
> /tmp/reproduced-datasets/supercomputer.csv
# Rexecute
```

[7] http://csvkit.readthedocs.org/en/0.9.1/

[8] www.top500.org

[9] https://eclipse.org/jgit/

[10] Code snippet: https://gist.github.com/stefanproell/b38e496a1259472c75f0

[11] Code snippet: https://gist.github.com/stefanproell/34f8ac3fb5b63599976f

```
/usr/bin/Rscript supercomputing/top5-script.r \
/tmp/reproduced-datasets/supercomputer.csv \
/tmp/reproduced-datasets/supercomputer-top5.csv
```

The method we propose is a simple way of storing reproducible data sets within Git repositories. The format of the metadata file serves as documentation and is machine actionable, as it allows retrieving the subset by executing the script file. The metadata can be parsed and used in a landing page, for increasing the readability for human users. It works well for simple scripts, which are not depending on processing chains with user interactions. It is designed to support one user per time per data set and implements an evolution pattern for each data set.

Note that in order for this approach to work, the repository has to ensure that the access/scripting language used to identify the subset is maintained. We thus recommend to only support subsetting functionality with a clearly and unambiguously defined semantic. All complex processing (e.g. data analysis, visualisation, etc.) should happen in subsequent processing scripts to keep the complexity of the long-term stability manageable. Considering more complex scenarios blurs the border between reproducible data sets and process preservation.

In addition to the R-based (or, in fact, any similarly structured script-like interface) we also provide support for subsetting using an SQL-like query language that can be executed against CSV files via the CSV2JDBC[12] library for Java, which allows retrieving subsets from CSV files via SQL statements. As both CSV and SQL are based on a tabular view of the data, CSV data can be easily mapped into a relational database table. Hence the translation process of a CSV subset selection process can be mapped to an SQL query. Figure 2 shows this transition.

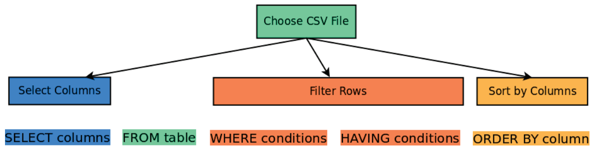


Figure 2: CSV Subsetting and SQL Queries

When a user wants to create an identifiable subset, we store the selected columns, the filter parameters and the sorting information in the query store. We preserve the SQL statement used for obtaining the subset in the first place. Additionally, we store the CSV file name and location and the execution timestamp in a metadata/landing page file, also stored in the Git based query store. As each metadata file has the unique PID as file name, the query can be re-executed based on the versioned CSV data set.

4.2 Using Git Branching to Separate Data and Queries

In Section 4.1 we introduced an approach for storing CSV data and metadata files in different folders in a Git repository. Furthermore we explained how to retrieve the metadata and CSV data files in order to re-execute the queries on the subsetted data. In this Section we will present a second approach that brings several advantages in a collaborative work environment.

When working with Git in a shared environment the concept of branching is the recommended best practise for allowing multiple researchers to work with different states of the data or files at the same time. A branch allows researchers to work with a specific version of data (or files) without distracting others. After the work has been completed (e.g. a subset has been created), the data can be merged with the main line or other branches again. At a certain point these branches are then merged together to a single branch to generate a common state.

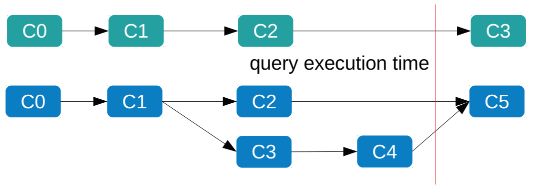


Figure 3: Commit graph without and with branches

Figure 3 shows two commit graphs. The upper graph represents commits to the repository done on a single branch, as described in Section 4.1. The graph below represents a repository were after commit `C1` a second branch was opened. The subsequent commits `C3` and `C4` were committed to the second branch. Commit `C5` is a merge commit where the two branches are merged together to a single branch.

If a query was executed at the time, that is marked by the red arrows in Figure 3, the algorithm introduced in Section 4.1 works differently on the two graphs, if it is re-executed at a time after commit `C5`. In the repository represented by the upper graph the algorithm returns the correct commit `C2`. In the repository represented by the lower graph the query would return commit `C4` instead of `C2` because it has a later commit date.

To solve this issue we need to change two aspects of the prior solution. First we need to save the CSV and metadata files in two separated branches instead of different directories. Second we change the algorithm to retrieve the data based on the timestamp to an approach were the specific commit hash is used.

If the CSV and metadata files are saved in the same branch, as in the approach described above, the history of CSV commits would be cluttered by the commits of the metadata. We therefore create a dedicated branch for the data and the queries[13].

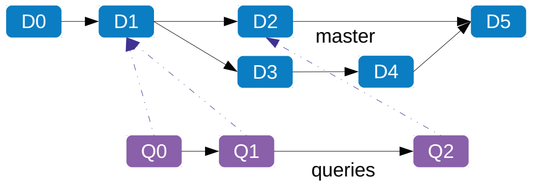


Figure 4: CSV(*master*) and metadata branch (*queries*)

Usually branches in Git share a common commit as ancestor, which means that the branches also share the same

[12] http://csvjdbc.sourceforge.net/

[13] https://gist.github.com/Mercynary/cc0d6bee550b701ab3c2ae6add75aee8

history up to this point. In Figure 3 the common ancestor is labeled as `C1`. Git also supports orphaned branches, where the the diverging branch gets a new commit as starting point for the history. The commit graphs of a repository with this configuration are displayed in Figure 4. In order to clearly separate the branches and their history, we create the query branch as an orphaned branch.

With the two branches created we can now change the algorithm to store the metadata files and retrieve the datasets on which the queries are executed on[14]. Figure 4 should thereby serve as an example of a repository with a *master* and a *queries* branch, as well as a branch that contains two CSV file commits on a third branch that was merged at a point later in time than the query commit `Q2`. In the figure the commits `Q1` and `Q2` refer to the CSV file commit `D1` as expected. However, the query commit `Q2` refers to the CSV file commit `D2` as the third branch was not visible to the application at the time of the query execution. In the description below the labels of the commits also represent the hash values of the commits.

Firstly, for saving a metadata file in the repository, the PID provided by the user is hashed with *SHA-1* to a string that can be used as a file name. We do this because PIDs could contain special characters, that are not permitted in a file name. Although we are aware of the probability of hash collisions, we chose *SHA-1* because Git uses the same algorithm to calculate the hashes of the committed files and thus limits our approach in the same way.

Secondly the previously created query branch is checked out and the contents of the metadata file are written. The most important information saved in the metadata is the query and its parameters, the data file name, the PID and the commit hash of the latest revision of the data branch. The commit hash is sufficient to identify the commit and its commit time in Git as well as to locate it on a specific branch if necessary, as the hashes remain the same when the branches are merged at a later point in time. In case of Figure 4 the saved commit hash in the metadata file would be `D2`. In a last step the metadata file itself is committed to the query branch which results in commit `Q2`. In this way the structure and history of the CSV data branch and the metadata branch do not interfere with each other.

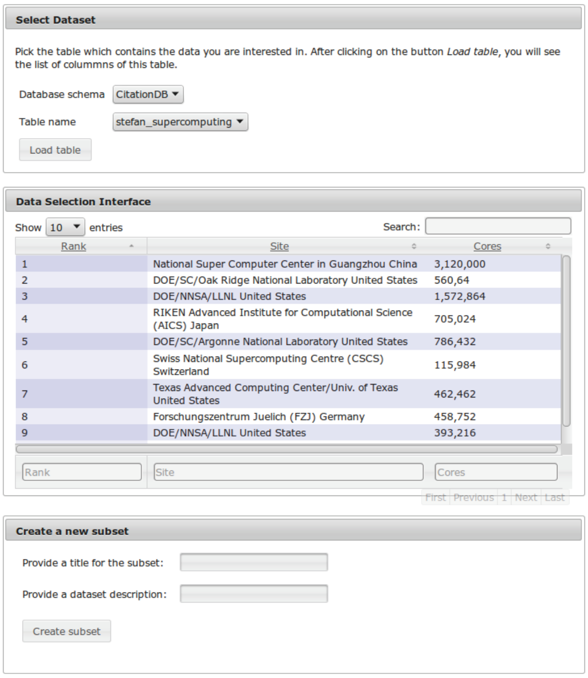
To retrieve the queries and re-execute them on the correct dataset, the following algorithm needs to be applied[15]. The user first provides the PID via the web application which is then hashed to get the file name of the metadata file. The next step is to checkout the query branch and read the metadata file identified by the hashed PID. From this metadata the commit hash and file name of the CSV data file can be extracted. In the example depicted in Figure 4 we would get the hash value `D2`. We then checkout the exact commit that is identified by the hash value. This way we restore the CSV data file as it was at the time the query was executed the first time. At this point we then can re-execute the query on the dataset.

Because the metadata files store the unique commit hash of the CSV data file in the repository, at the time when the query is stored and executed, the commits can not get mixed up when two or more dataset branches are merged together in advance to the approach that was based on timestamps. As mentioned in the beginning of this section, due to the solution of separated CSV data and matadata file branches as well as a metadata retrieval based on commit hashes this approach is better suited when working in a collaborative environment. We implemented a prototypical web application[16] as a proof of concept.

4.3 Reproducible Subsets Based on Migration and Database Queries

One major disadvantage of CSV files is the lack of native support for subsetting, i.e. selecting only a specific set of rows and columns. While the Git approach is suitable for smaller scale files, native support is preferable for larger data files, allowing to extract only smaller files from a repository in the first place, rather than having to extract all data and performing the subsetting afterwards.

Our third implementation still transparently provides CSV files for the researcher, but internally utilises the advantages of a database management system which takes care of versioning the data. Users upload their CSV files into a Java application and they can generate a subset by using a Web interface. Subsets can be retrieved again as CSV file download, where the files are created on demand based on queries. We implemented a two phased migration process for inserting the data into a MySQL 5.7 database management system. Figure 5 shows the interface of our prototype solution with three selected columns.



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 colummns of this table.

Database schema: CitationDB

Table name: stefan_supercomputing

Data Selection Interface

Show 10 entries

Rank	Site	Cores
1	National Super Computer Center in Guangzhou China	3,120,000
2	DOE/SC/Oak Ridge National Laboratory United States	560,64
3	DOE/NNSA/LLNL United States	1,572,864
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	705,024
5	DOE/SC/Argonne National Laboratory United States	786,432
6	Swiss National Supercomputing Centre (CSCS) Switzerland	115,984
7	Texas Advanced Computing Center/Univ. of Texas United States	462,462
8	Forschungszentrum Juelich (FZJ) Germany	458,752
9	DOE/NNSA/LLNL United States	393,216

Create a new subset

Provide a title for the subset:

Provide a dataset description:

Figure 5: An Interface for Creating Reproducible Subsets

In the first phase, the CSV file parsed and a table schema based on the file structure is created. CSV header names (i.e. the first row in the CSV file) serve as column names

[14] https://gist.github.com/Mercynary/bac394c035b0a98b338202d3e8705768

[15] https://gist.github.com/Mercynary/8703ea098bcec52a2233537aaecc9f20

[16] https://github.com/Mercynary/recitable

for the table. In cases where a CSV schema[17] file is available, the data type can be specified for the columns within the database table. If no schema is available, the data in each column can be analysed and heuristics can be used to determine an appropriate data type (date, numeric, string, etc.), or all columns can simply be interpreted as text strings (VARCHAR columns). By parsing the file once, columns containing potential identifiers can be detected. We use these identifiers as primary keys for differentiating between records. If no candidate is available, this can either be an indicator for duplicate records in a CSV file or the set simply does not provide unique columns which could serve as identifiers, a sequence number column generated automatically by the system is appended to the data set for internal use. Each newly generated table is expanded by one column for timestamps and one column for storing the event type (*INSERTED*, *UPDATED*, *DELETED*). Having the two additional columns available allows implementing a versioning scheme as described in [18]. In the second phase, the CSV file is read row by row and the data is inserted into the database table. For each newly added record, the system automatically inserts the timestamp and marks the data as inserted.

For adding data to the set, users can provide CSV files with the same structure and upload them into the system. Header names can serve for checking whether the file still has the same structure and the column type heuristics can be applied for checking if the column type remained the same. During the upload, the file is parsed and the records are inserted into the data set, where the primary key column defined in the database ensures that updates can be detected.

Upon the upload of a file containing changes, old records are marked as updated and the updated version of that record is inserted as a new version with the current timestamp and the *INSERTED* mark. Obviously, detecting which record to update only works if a primary key is present in the updating file. In case where no such unique column is available, researchers can download the current version of the data set including the sequence number column. By updating this file, for instance by using some spreadsheet software, the sequence number can be preserved and used as a matching column.

The query store is implemented as a separate database schema, providing tables for storing the metadata for retrieving the queries at a later point in time. The query metadata includes source table, query parameters, execution time and the persistent identifiers assigned to the query. As soon as the data has been migrated into the RDBMS, the advantages of the query based mechanism can be used for identifying specific subsets of research data. This allows to re-execute the query and map the timestamp of the query execution time against the versioned data set. The subset which is defined by the information stored in the query store can then be retrieved on demand.

5. EVALUATION OF THE DATA CITATION APPROACHES FOR LONG TAIL RESEARCH DATA

In this paper we presented three approaches for creating reproducible and citable subsets of CSV data. All three are based on versioned and timestamped data and utilise a query mechanism which retrieves the data at it was at a specific point in time.

5.1 Using Git for Versioning, Identification and Re-Execution

Versioning data sets with Git is easy to integrate and commonly recognised as good practice for text based data formats. The overhead created by the Git repository is low and does not require sophisticated server infrastructure. Interpreting software scripts as query allows to create subsets in a flexible way. Instead of adding subsets directly into the Git repository as new files, the query string or script can be used for retrieving the data from the versioned data set. The query or scripts respectively are versioned as well and thus can be mapped to a specific version of a subset. As the version of the data set can be obtained from the repository, the likewise versioned query can be re-executed without any modifications. The mechanism can be applied to any scripting language, as long as the required commands and parameters are stored in the query store.

By adding the script files responsible for creating the subsets under version control, internal mechanisms of Git can be used to re-execute the subset creation process at any timestamp. Various versions of the script can be applied on the available history of data sets. This does not only enable reproducibility, but also allows to compare different versions of the subsetting process with each other.

The required software is open source and thus freely available and used by a very large community. The Git based approaches can therefore easily be implemented in long tail data settings. Furthermore it can be integrated into existing processing pipelines, adding reproducibility for the data input and output processing steps.

Git utilises a line based approach for interpreting differences in versions of data. Thus the traceability of changes between two versions is limited, if the granularity is below row level. Sorting for instance can hardly be differentiated from updating records, which results in the deletion and subsequent addition of a record into the file.

Re-ordering a CSV file by changing the sequence of columns also leads to a completely different file, as all of the records are considered as deleted and new records are detected to be added. For this reason different versions of one data set cannot be compared reliably without additional tools, leading to less-efficient utilisation of storage. On the other hand, as CSV files tend to be moderately-sized, this does not constitute a major limitation.

Similarly, for retrieving a subset, the entire CSV file first has to be checked out of the repository before the appropriate subset can be extracted by running the original script. While this might be undesirable in massive-scale data settings it is unlikely to cause major problems in typical settings employing CSV files.

These limitations of the Git based approaches are due to the focus of Git on source code rather than data files. The Git approaches allow utilising one single versioning system for both, code and data. Therefore, no complex infrastructure or maintenance is required and the integration of the data citation solution into existing workflows suitable for any kind of ASCII data files and scripting languages for

[17] http://digital-preservation.github.io/csv-schema/csv-schema-1.0.html

retrieving the subsets requires low overhead. Subsets can be compared across different versions by creating delta files (also known as diffs) and the differences can be visualised or extracted.

5.2 Using Database Systems for Versioning, Identification and Re-Execution

Advanced database technologies support very large data sets and provide a higher performance than the file based approaches. Flexible query languages such as SQL allow retrieving specific subsets from data. Using a graphical interface hides the complexity of the query language and users can select and re-order columns in the data set, filter and sort the rows according to specific criteria, much as they are used to work with data in spreadsheet programs.

Rewriting the queries for retrieving the version valid at a specific timestamp is a necessity, but can be automated by intercepting the commands from the interface. By using triggers all of the required operations can be automated and indices on the metadata columns increases the query performance for versioned data. In our approach, we store the filter and sorting criteria from these queries in the query store. Thus we can re-execute any query at any available point in time on the versioned data and provide additional services on top such as result set verification by hashing.

The database approach overcomes the limitations of the Git based data citation approach as it does not rely on line level versioning, but allows fine granular data citation even for single cells in a table. As SQL is a declarative language, the subsets of data are described in an abstract way, which allows domain experts to understand what a specific query returned as result. This information can be represented in different forms and can be reused for instance for providing automated citation text snippets automatically.

Subsets can be compared across different versions, simply by re-executing the stored query with different timestamps. Differences can be made visible by comparing the returned result sets and exporting the differences. Handling alternative sortings or a different sequence of the columns of a data set can be easily handled by rewriting queries, without the need of changing the underlying data set.

The flexibility offered by this approach comes with the cost of increased complexity. The data needs to be imported into the database system, which is responsible for versioning both, the data and the queries including their metadata. Also the interface needs to support users depending on the requirements of the domain, therefore the solutions may often not be applicable as a generic tool across community boundaries.

5.3 Preserving the Subset Creation Process for the Long Term

Both approaches for data citation allow researchers to generate precisely defined subsets from CSV files. The processes we described provide reproducibility for data sets, as they tie the versioned data and a timestamped query together. In contrast to storing the different versions of a subset as individual files, the processes require less disk space and the information how a subset was created is maintained inherently, as this information is contained in the query. This allows domain experts to understand what records have been included into a subset and which ones have been omitted. As all parameters of the query are stored, either in the query store explicitly or within a script, the subset creation process can be understood ex post. This knowledge contributes not only for the long term reproducibility, but also adds valuable metadata automatically, as the intention of a subset can be traced even if the data should be long gone.

While both approaches are simple and easy to implement, they both rely on the availability of the query language (i.e. SQL, software scripts, etc.) and the execution environment of the query engine. In order to keep these subsets accessible, the environment and the query engine need to be preserved for the long term. As technology progresses, the emulation of the original environment or the migration to a new environment may become necessary. Our approach is based on two de-facto industry standards: Git and SQL. For source code versioning, migration pathways to new versions of the Git software or other source code management systems exist already today. The same applies to the database system MySQL, which offers many migration pathways to other (relational) database systems and even back to CSV. Whenever one of the two system needs to be moved to a virtualised environment or migrated into a different environment, the correctness of the re-executed subsets needs to be verified. This can be achieved by comparing the hash values from the old and the new system.

6. OUTLOOK AND CONCLUSIONS

In this paper we present three methods for the precise identification of arbitrary subsets of CSV files even when these data files are evolving over time. The three methods have in common that they allow to make specific subsets of data citable, by assigning a PID to the subsetting process ("query") executed against a specific state (timestamp) of the versioned data source. Additionally, we store the query or script respectively, which created the subset in a versioned fashion. We establish a link between the versioned data set at a specific time and the query as it was executed at that point in time. Being able to reproduce the process of subset creation allows us to shift the identification mechanism from data set level to the query level. This produces much less storage overhead as the duplication of data is avoided. Storing query metadata does not require significant storage compared to versioned subsets of data.

The solutions we propose have been developed with a focus on simplicity, low overhead, low maintenance and the ease of use in various research settings. The steps necessary to create citable subsets can be fully automated, relieving the researcher from the burden of individual data management, i.e. manually maintaining multiple copies of data files. The approaches can be used in combination with a centralised repository or individually at the researchers work station.

The first two approaches rely on an underlying Git repository to be used for data storage and for providing versioning capabilities in long tail research data settings. The subsetting is performed by scripts which create a subset from a data set. Both, the data and also the scripts required for creating the subset are stored in a Git repository. Additional metadata allows to re-create a subset as it was at any given point in time. Descriptive information can be added, which allows human beings to understand how a subset was created which further improves the reproducibility of data driven experiments. The first approach is simplistic, equipping researchers with a simple yet powerful method for creating citable data sets, by storing the data and the script in a dedicated repository in a linear fashion. Each subset becomes identifiable with a PID. The second approach adds parallelism to the approach and allows several researchers to simultaneously work with data sets without distracting each other. The results can be compared and easily shared.

In the third implementation, the CSV data is migrated into a relational database. Subsets can be generated either directly via an API accepting SQL queries, or via a graphical interface mimicking a spreadsheet program. By storing the data as well as the subsetting information in a versioned fashion in a database system, subsets from very large data sets can be made citable in a efficient way. Additionally, the proposed methods allow comparing different versions of the same subset more easily and allow generating subsets with the same characteristics also from newly added data. Storing the query allows retrieving in fact any subset version of evolving data and enhances the reproducibility of data driven research in larger scale settings.

Acknowledgement

Part of this work was co-funded by the research project DEXHELPP, supported by BMVIT, BMWFW and the state of Vienna, and COMET K1, FFG - Austrian Research Promotion Agency.

7. REFERENCES

[1] Juha K Laurila, Daniel Gatica-Perez, Imad Aad, Olivier Bornet, Trinh-Minh-Tri Do, Olivier Dousse, Julien Eberle, Markus Miettinen, et al. The mobile data challenge: Big data for mobile computing research. In *Pervasive Computing*, 2012.

[2] Derek John de Solla Price, Derek John de Solla Price, Derek John de Solla Price, and Derek John de Solla Price. *Little science, big science... and beyond*. Columbia University Press New York, 1986.

[3] Christine L Borgman, Jillian C Wallis, and Noel Enyedy. Little science confronts the data deluge: habitat ecology, embedded sensor networks, and digital libraries. *International Journal on Digital Libraries*, 7(1-2):17–30, 2007.

[4] Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. Ten simple rules for reproducible computational research. *PLoS Comput Biol*, 9(10):e1003285, 10 2013.

[5] Andreas Rauber, Ari Asmi, Dieter van Uytvanck, and Stefan Proell. Data Citation of Evolving Data - Recommendations of the Working Group on Data Citation. https://rd-alliance.org/rda-wgdc-recommendations-vers-sep-24-2015.html, September 2015. Draft - Request for Comments.

[6] Andreas Rauber, Ari Asmi, Dieter van Uytvanck, and Stefan Proell. Identification of Reproducible Subsets for Data Citation, Sharing and Re-Use. *Bulletin of IEEE Technical Committee on Digital Libraries*, 2016. Accepted for Publication. URL: https://rd-alliance.org/system/files/documents/RDA-Guidelines_TCDL_draft.pdf.

[7] Heather A Piwowar and Todd J Vision. Data reuse and the open data citation advantage. *PeerJ*, 1:e175, 2013.

[8] Mark P Newton, Hailey Mooney, and Michael Witt. A description of data citation instructions in style guides. 2010.

[9] Jochen Kothe Hans-Werner Hilse. *Implementing Persistent Identifiers: Overview of concepts, guidelines and recommendations*. Consortium of European Research Libraries, London, 2006.

[10] Ruth E Duerr, Robert R Downs, Curt Tilmes, Bruce Barkstrom, W Christopher Lenhardt, Joseph Glassy, Luis E Bermudez, and Peter Slaughter. On the utility of identification schemes for digital earth science data: an assessment and recommendations. *Earth Science Informatics*, 4(3):139–160, 2011.

[11] Norman Paskin. Digital Object Identifier (DOI) System. *Encyclopedia of library and information sciences*, 3:1586–1592, 2010.

[12] Tobias Weigel, Michael Lautenschlager, Frank Toussaint, and Stephan Kindermann. A framework for extended persistent identification of scientific assets. *Data Science Journal*, 12(0):10–22, 2013.

[13] John F Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

[14] R. Chatterjee, G. Arun, S. Agarwal, B. Speckhard, and R. Vasudevan. Using data versioning in database application development. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 315–325, May 2004.

[15] Divyakant Agrawal, Amr El Abbadi, Shyam Antony, and Sudipto Das. Data management challenges in cloud computing infrastructures. In *Databases in Networked Information Systems*, pages 1–10. Springer, 2010.

[16] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang-Chiew Tan. Archiving scientific data. *ACM Trans. Database Syst.*, 29(1):2–42, March 2004.

[17] Stefan Pröll and Andreas 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.

[18] Stefan Proell and Andreas 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.

[19] Stefan Pröll and Andreas 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.

[20] Matthias Schwab, Martin Karrenbach, and Jon Claerbout. Making scientific computations reproducible. *Computing in Science & Engineering*, 2(6):61–67, 2000.