# Identification of Reproducible Subsets for Data Citation, Sharing and Re-Use

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

Ari Asmi
University of Helsinki, Finland
ari.asmi@helsinki.fi

Dieter van Uytvanck
CLARIN ERIC, Utrecht, Netherlands
dieter@clarin.eu

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

## ABSTRACT

Research data is changing over time as new records are added, errors are corrected and obsolete records are deleted from a data set. Scholars rarely use an entire data set or stream data as it is, but rather select specific subsets tailored to their research questions. In order to keep such experiments reproducible and to share and cite the particular data used in a study, researchers need means of identifying the exact version of a subset as it was used during a specific execution of a workflow, even if the data source is continuously evolving. In this paper we present 14 recommendations on how to adapt a data source for providing identifiable subsets for the long term, elaborated by the RDA Working Group on Dynamic Data Citation (WGDC). The proposed solution is based upon versioned data, timestamping and a query based subsetting mechanism. We provide a detailed discussion of the recommendations, the rationale behind them, and give examples of how to implement them.

## 1. INTRODUCTION

In data driven experiments, the data processed is not always static, but rather evolves based on collaborative work and iterative improvements. Researchers contribute collaboratively to data sets by providing records created during their experiments. In addition to newly added data, corrections of detected errors as well as deletion of outdated information account for the dynamics of research data sets.

Data sets serve as input for processing steps, thus an update or change within a data set causes variation in the results downstream the processing pipeline. Tracing the specific version of a data set or subset is therefore essential for being able to verify a specific outcome. This verification is essential for measuring the success of an experiment and for the scientific method in general, as it allows peers to repeat an experiment and assess its validity. This process is denoted as reproducibility and it deals with the verification of experiments and their results. Being able to reproduce an experiment involves not only precise knowledge of all processing steps, but also the availability of exactly the subset of data used during a specific experiment run. It further is essential to support comparability of different methods and approaches being evaluated on identical data.

Researchers often create subsets specifically tailored for an experiment setup. Such subsets are usually obtained by issuing some form of query which returns only the records desired. Obviously, storing all revisions of subsets of evolving data as separate data exports does not scale with increasing amounts and volumes of data. For this reason, we introduce a query based subset identification method, which overcomes the duplication of data exports and provides lightweight yet precise subset identification for evolving data.

In this paper, we present 14 recommendations for creating reproducible subsets elaborated by the Working Group on Dynamic Data Citation[1] (WGDC) of the Research Data Alliance (RDA) [23]. They have been improved in several rounds of expert feedback and conceptual validation in the context of a range of domain-specific data centres. The recommendations serve as guideline how to allow identifying dynamic subsets of data from existing data sources. They enable researchers and data centres to identify and cite data used in experiments and studies. Instead of providing static data exports or textual descriptions of data subsets, our solution supports a dynamic, query-centric view of data sets. It is based upon assigning PIDs to time-stamped queries which can be re-executed against a versioned database. The proposed solution enables precise identification of the very subset in the correct version, supporting the reproducibility of processes, sharing and reuse of data. It is generally applicable for different types of data, data stores and subsetting paradigms (such as CSV files, SQL, XML, streaming data, file repositories and others).

So far, persistent identifiers (PIDs) have mainly been used for identifying individual objects in a static context. The solution proposed in this paper allows accessing individual subsets of data in a dynamic context, supporting the identification of fine granular subsets of evolving data. By assigning PIDs to the query, the process is very lightweight and scales

[1]`https://www.rd-alliance.org/group/data-citation-wg.html`

with increasing amounts of data. It preserves the subset creation process and thus contributes to the reproducibility of an experiment. Provenance details and metadata about the data set are collected.

The remainder of this paper is structured as follows. Section 2 provides related work in the areas of reproducibility, data citation and databases. The recommendations are explained in Section 3 detailing our considerations, existing challenges and an example for each recommendation. The paper is closed by conclusions in Section 4.

## 2. RELATED WORK

A scientific process is reproducible if an independent peer scientist can re-execute the experiment and validate the results [18]. Currently, there is a lack of reproducibility and many workflows cannot be re-executed [28, 17, 13], although recent initiatives aim to demand the reproducibility of submitted contributions [16, 9]. Being able to precisely reference data sets is the goal of data citation as it supports reproducibility [12]. The field of data citation as evolved quickly [3], an overview of the current practices for citing data is given in [8]. Several approaches exist for citing subsets from databases [20], structured data files [21] or linked and open data [26].

Versioning data is a common task in the data management domain [24] and implemented in software applications dealing with critical data [7, 25]. With decreasing storage costs preserving previous versions even of high volume data has become a service offered by many data providers. Still storing multiple versions is a challenge [1]. Many efforts of storing previous versions of datasets are based on timestamps [5] and temporal data models [27]. The ISO SQL:2011 standard includes temporal features such as system versioned tables [15]. Recently, many relational database management systems (RDBMS) such as MS SQL Server, Oracle 10g or DB2 10 support versioned data natively and allow querying data as it was at any given point in time in the past.

## 3. ENABLE REPRODUCIBLE DATA CITATION OF DYNAMIC DATA SOURCES

The Research Data Alliance (RDA) is an international community of researchers from diverse backgrounds aiming to overcome barriers to sharing research data across international, institutional and disciplinary boundaries. RDA was launched in 2011 and spans private, public, academic communities from all continents. It is an international non-profit community of researchers, which is organised in interest groups (IG) and working groups (WG). The RDA Working Group on Data Citation (WGDC) aims at bringing together experts to discuss the issues, requirements, advantages and shortcomings of existing approaches for efficiently citing subsets of data.

### 3.1 RDA Guidelines for Making Data Citable

During the course of its existence, the WGDC developed the RDA Guidelines for Making Data Citable, which is a list of 14 recommendations grouped in four areas:

- Preparing the Data and the Query Store
  - R1 - Data Versioning
  - R2 - Timestamping
  - R3 - Query Store Facilities
- Persistently Identifying Specific Data Sets
  - R4 - Query Uniqueness
  - R5 - Stable Sorting
  - R6 - Result Set Verification
  - R7 - Query Timestamping
  - R8 - Query PID
  - R9 - Store the Query
  - R10 - Automated Citation Texts
- Resolving PIDs and Retrieving the Data
  - R11 - Landing Page
  - R12 - Machine Actionability
- Upon modifications to the Data Infrastructure
  - R13 - Technology Migration
  - R14 - Migration Verification

The guidelines have been finalised at the 6th Plenary Meeting in Paris in September 2015. They have been published as a short flyer [23] and are available for download at the RDA Web page[2].

The aim of this work is to provide a more extensive discussion of the guidelines and describe how an existing data source can be adapted for enabling reproducible subset creation. The four phases are described in the following sections: preparing the system (Section 3.2), persistent identification (Section 3.3), subset retrieval (Section 3.4) and robustness across technological changes (Section 3.5).

### 3.2 Preparing the System

Although some data repositories and storage systems already have some of the requirements in place, most systems need to be adapted for facilitating reproducible data set creation in a non-intrusive way.

#### *3.2.1 R1 - Data Versioning*

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

Versioning deals with tracing changes in data. Being able to refer to previous versions of data is essential for reproducibility. In this work, we use the terms version, revision and iteration synonymous for describing a change which was introduced into a record. There exist several approaches for versioning from the perspective of reproducible research data. In order for the changes in the data to be traceable, the system needs to store all relevant previous versions of each record. In large data scenarios, storing all revisions of each record might not be a valid approach. Therefore in our framework, we define a record to be *relevant* in terms of

[2]`www.rd-alliance.org/groups/data-citation-wg.html`

reproducibility, if and only if it has been accessed and used in a data set. Thus, high-frequency updates that were not ever read might go - from a data citation perspective - unversioned. (They may still require versioning if traceability of changes and analysis of update frequencies are to be supported by the data store). Note, that the versioning needs to be modelled explicitly so that direct access to earlier states is possible. History files or change logs that may be used to roll back an entire database to an earlier state are usually not flexible enough to support retrieving individual states of specific records.

*Example*: Deciding which versioning approach to apply depends on the format of the data and the expressiveness of the query language. There exists a huge variety of data set types, ranging from simple plain text formats such as comma separated value (CSV) files, via complex binary proprietary formats, to various database management systems. Research data can be stored as files within a file system or in database management systems. File systems provide a functionality similarly to database systems, as files can be uniquely addressed by their path in the file system and queried using file system commands.

In order to provide a uniform way of describing events, we limit ourselves to the three most primitive data operations available for all types of data.: *create*, *update* and *delete* (CUD). These event types are sufficient for describing the life cycle of a record. The create event denotes the insertion of a record into the data set and marks the beginning of a life cycle. Depending on the scenario, records can get updated and therefore changed, thus single records can be replaced with a newer version. Such updates or deletes should never lead to an actual deletion or overwriting of an earlier record, but simply lead to a marked-as-deleted and re-insertion of an updated value in the underlying data store.

In the realm of databases, versioning has a long tradition [14]. Approaches to support versioning include history tables, direct integration in the live database tables, or hybrid solutions [19]. Following the ISO SQL:2011 standard, many RDBMS such as MS SQL, Oracle or DB2 support it now natively. Managing different versions of text files (such as data stored in CSV files) also has a long tradition in the area of software development. Source code management (SCM) systems such as Git, SVN or CVS are commonly used for storing source and allow programmers to access previous versions of their source code. The technique of storing the deltas of files works very well for text based formats.

### 3.2.2 R2 - Timestamping

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

Timestamping is closely tied to versioning: we need to identify, when a specific version was available (visible to a query). We thus need to timestamp each CUD operation as part of the versioning approach, documenting the addition and deletion of data. For providing reproducibility, an update leads to a new record in the data set, where the old revision of a record is stored with the temporal metadata describing its validity period in the database. Timestamp information must be stored in a way to support efficient explicit querying, i.e. allowing a query to retrieve only data elements available at a given point in time.

*Example*: Database management systems are capable of storing fractions of seconds such as micro seconds (MySQL 5.7, PostgreSQL 9.4), nano seconds (Oracle 10g) and even pico seconds (IBM DB2 11.0) as dedicated data types. Distributed systems can synchronise their timing either via network time synchronisation and logical time protocols [11], but may also work with local timestamps only (c.f. Sec. 3.3.4), which drastically reduces the complexity of the recommended solution.

The way how timestamps are implemented depends on the technology stack used for storing the data. File systems such as FAT provide two second accuracy[3], NTFS provides 100 nano seconds resolution[4], EXT3 provides 1 second accuracy and EXT4 allows nano second resolution of timestamps[5]. The file system accuracy of timestamps also determines the accuracy of the version control system timestamps, such as for Git or SVN.

### 3.2.3 R3 - Query Store Facilities

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

The query store is an essential building block for the dynamic identification of data subsets. It was introduced in [19], refined in [20] and generalised in [22]. Instead of storing duplicates of data subsets, we utilise query mechanisms in order to define and identify subsets of data. To identify a particular subset of data, users issue some form of query to a data store (e.g. file system commands, SQL or SPARQL queries, faceted browsing or filtering, marking regions on an image, etc). These queries can be timestamped and stored for later re-execution against the timestamped and versioned data store. The timestamped query thus stands for the actual subset of data.

The query store is itself a repository responsible for storing the queries which were used for creating a subset. This includes technical and descriptive metadata, which allows to describe a query in the detail necessary to re-execute it against versioned and timestamped data and retrieve the very same subset again. The query store needs to preserve this information for the long term. It needs to store the query and at least the set of metadata information emerging from recommendations *R4-R10* as detailed further below. This set includes:

[3]https://msdn.microsoft.com/en-us/library/windows/desktop/ms724290%28v=vs.85%29.aspx
[4]http://www.meridiandiscovery.com/articles/date-forgery-analysis-timestamp-resolution/
[5]http://www.ibm.com/developerworks/linux/library/l-anatomy-ext4/

- The original query as posed to the database
- A potentially re-written query created by the system (R4, R5)
- Hash of the query to detect duplicate queries (R4)
- Hash of the result set (R6)
- Query execution timestamp (R7)
- Persistent identifier of the data source
- Persistent identifier for the query (R8)
- Other metadata (e.g. author or creator information) required by the landing page (R11)

The query store preserves details about the creation process of each subset, including the relevant parameters, their sequence of occurrence and settings, user details, timestamps and verification information. This constitutes a valuable provenance trail and increases trust in the scientific process, as the metadata about subsets can outlive the original data sets and still provide details about the subset generation process.

It also provides a valuable central source for analysing and understanding data usage, i.e. which parts of a data store are used, while allowing also to identify those subsets used previously that are affected by e.g. a specific update to the data. This is also the reason why the query store is a sensitive infrastructure and needs to be protected from unauthorised access. It is clear that the data stored in the query store must not be changed or manipulated, but also the provenance data about the queries needs to be protected from unauthorised read access. Knowing how a researcher created a subset may disclose private information about the current research practices and interests of a researcher.

*Example*: The way how query execution details are stored in the query store depends on the used technology. Subsets are created by executing SQL statements, extracting data from a file based data set by using a scripting language or selecting files from a file system are examples for queries. At the end, all data stores can be seen as a database, where a query method allows to programmatically retrieve data. Thus the query store either needs to preserve the database query, the script used for retrieving the data or the path instructions used for retrieving a subset in a versioned fashion and link this information in the query store with the data source (i.e. the file, database, etc). For retrieving such a subset again, the data needs to be collected from the versioned data source and the appropriated version of the script needs to be re-executed. For the result set to be identical to the original subset, the query store also needs to preserve which version of the scripting interpreter was used, including the data of the execution environment, such as operating system and used libraries. This topic is borderlining the area of process preservation.

## 3.3 Persistently Identify Specific Data Sets

Storing each and every query to a data source may not be required or desirable. The system or users may want to decide which queries should be preserved. The process can be limited to specific data sources, be triggered by a manual switch in a user interface or any query can be stored in a staging area for a certain period of time, within which a final decision on the usefulness and subsequent usage of a dataset has to be made, upon which the persistent storage can be initiated. In any case, when a specific subset should be made persistently available, the following recommendations should be observed.

### *3.3.1 R4 - Query Uniqueness*

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

Semantically identical subsets should be identified by only one PID. Thus, we need to identify, if the same query has already been issued before (and if so, whether it still identifies the same subset of data, depending on any updates to the data source since an earlier execution).

Determining the identity or uniqueness of queries may be challenging as the same query semantics may be expressed in many different ways. In order to detect duplicate queries, we need to normalise the queries. Many command line applications, for instance, allow specifying the input parameters in arbitrary sequence. The same applies for database query languages, which provide additional flexibility. We thus need to normalise queries as far as possible, to allow detecting semantically equivalent queries.

While it may be impossible to guarantee detection of identical queries as the same semantics may be expressed in a variety of forms, the situation is pragmatically eased by the fact that users will frequently not actively write queries but use standardised interfaces (e.g. for faceted browsing) to create queries. This also applies to queries posed via APIs that will frequently have been produced programmatically, thus ensuring a certain amount of standardisation and facilitating the detection of semantically equivalent queries.

A rather simple solution is to sort the parameters in a static way for detecting duplicates. We sort the list of parameter name and value pairs alphabetically and utilise the concatenated string as input for a hash function. Thus we can detect duplicate queries quickly and reliably for many applications. However, if two semantically equivalent queries are formulated so differently that they evade detection, we may end up in the worst case with having two identical data subsets identified by two different PIDs. While this may not be ideal, it does not constitute major harm either. Note, also, that the fact that two result sets are identical, can be detected automatically (cf. R6) thus alerting the user to potentially identical queries. Mind, however, that identical result sets do not necessarily mean that the underlying queries are semantically equivalent, as discussed in Section 3.3.3.

### 3.3.2 R5 - Stable Sorting

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

The sequence of records within a set can have an effect on the result of an experiment, if the subsequent processing is sensitive to the order the input data is provided in. Yet, while databases and repositories will return the same result set when repeating a query, these are not necessarily always returned in the same order. We may thus need to ensure that the sequence or sorting of the records within a subset is unambiguous and reproducible itself.

Whether or not the sequence of records in a data set is deterministic depends on the storage system and the query language. A set in its mathematical definition does not maintain a sorting, therefore there is no defined sequence of records. As many database management systems are set based, there is no guarantee, that the records of a data set will be returned in the very same sorting for each request. For this reason, we may want to enforce an explicit sorting of the data prior to applying user-defined sorts.

*Example*: The popular RDBMS MySQL provides several different storage engines, such as InnoDB and MyISAM. Both differ in their features and how the records are stored on the disk. InnoDB has features such as transactions, foreign keys and clustered indices. The last feature has an effect on the sorting of the result set as InnoDB stores the records sorted by their index on disk, whereas MyISAM stores the records based on their insertion order. If the user does not explicitly define an indexed column by which the result set should be sorted, the sequence of the records cannot be predicted. As the storage engine can be altered at any time, without the knowledge of the users, the sequence of records may change for identical queries.

For this reason, the sorting of the result set should be specified. It needs to be based on a unique criterion, which allows creating a total ordering of the records in the data set and the subsets thereof. We can obtain a stable sorting by automatically specifying the primary key column in descending order as default sorting, followed by user defined sortings.

### 3.3.3 R6 - Result Set Verification

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

By correctness we mean that no record has (intentionally or unintentionally) been changed. By completeness we understand that all records, which have been contained in the original subset, are also contained in the re-generated subset. When a user creates a new, reproducible subset, we compute a hash key $k$ of this set and store it in the query store. This hash key allows to verify the completeness and correctness of the subset, by computing and comparing the hash key $k'$ of the re-generated subset to the original key $k$. Only if the keys are identical, the re-executed subset is correct. This hash key may also serve to identify identical result sets stemming from different queries.

Computing hash keys is a computationally intensive task. Thus, the computation of a large subset's hash key may require too much time to be practical in many situations. For this reason, the verification procedure can be sped up, by computing the hash key based only on the significant properties of a subset, instead of the complete data contained in it. Thus, the hash key can, for example, be computed only over the primary key and row headers to define the matrix of the result set. Another option is the computation of semantic hash keys [2] that were originally proposed for verifying the correct interpretation of data when moved between different systems.

Additional metadata information such as the timestamps, but also the data can be included. Note that, to guarantee long-term stability, some transformation to generate the hash input values from the result set may be required, e.g. removing certain object-specific formatting such as the separator in CSV file, unless the separator is considered essential for identity determination.

*Example*: For calculating the hash of a subset, the system retrieves the dataset by re-executing the query stored in the query store and the list of selected columns from the query store. The sequence of columns selected is concatenated into a input string for the hash function. In a next step, the system iterates over the result set. For each record contained in the result set, the system appends the unique key column and the timestamp of the record in a string. The hash key is thus computed over the row and column headers for tabular data. For file system based data, the hash key may be computed over the file IDs (PIDs, URIs, local path names) plus timestamp information when the respective version was added/modified.

### 3.3.4 R7 - Query Timestamping

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

Assigning the timestamp to a query can be as simple as assigning the time of the actual execution of the original query. This would constitute the simplest and most intuitive timestamp to be assigned. However, during discussions, privacy concerns have been raised as the timestamp reveals the moment when a query has been issued by a user, which may disclose unwanted information. On the other hand, semantically we would be more interested in referring conceptually to the actual state that the data store was in. We thus recommend assigning the timestamp of the last update to the data store, as this intuitively corresponds to the version number and timing information of the state of data. A third alternative is to provide the timestamp of the last update to the subset of data affected by the query, which may be most intuitive for data stores where changes are local to specific subsets (i.e. only new data being added, thus not affecting any queries for earlier time periods). While this constitutes a perfectly viable solution, determining this timestamp is a bit more complex, making it not necessarily the primary option. This method requires to retrieve the subset first and then determine the latest updates to this subset. Note that, whichever timestamp is chosen, the re-execution of the query with that timestamp will always return the same result.

Dealing with distributed data sources does not require global synchronisation, as each distributed source has its own local timestamp and can return this with the result to the central query portal. There, the query is stored with the central timestamp as well as the execution timestamps of each decentral site as returned with the result set.

*Example*: The query execution time can be mapped to the latest update of the database. This approach follows a global view on the data. More frequent updates suggest the implementation of a local timestamp, which is the most recent update of the records included in the selected subset, utilising a more local view on the data. Figure 1 depicts both approaches.

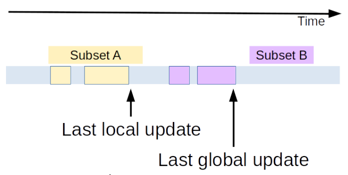


Figure 1: Assigning a Timestamp

### 3.3.5 R8 - Assigning Query PID

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

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. 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 3.3.1, 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 an identical query, which has already been issued before, there are two possibilities. Either the underlying data has not been changed, in which case the existing PID will be returned to the user as both query and result are identical. If the data has been updated between the last execution of the same query and the current execution time leading to a different result set, a new PID has to be assigned. If a new PID has been assigned, the query and associated metadata needs to be persisted (R9). The PID could consist of the hash key of the query plus timestamp information. Yet, we highly recommend adopting any of the established PID systems (DOIs, ARKs, or other URI-based systems) that are widely adopted within a specific community [10] to support interoperability, ease resolution via established protocols, and may even support machine-actionability via content negotiation, depending on the provider of the PID system.

*Example*: In order to decide, whether or not a new PID needs to be assigned, we check whether the query hash (*R4*) (e.g. SHA-1 checksum) does not yet exist in the query store. If the query hash does not exist, we immediately assign a new PID and proceed with the further steps. In case the query hash exists, we compute the SHA-1 checksum of the result set and compare it to the checksum stored for the original execution of the identical query. If the checksums match, both the query as well as the result set are identical, and we thus should not assign anew PID to this subset of data, but rather return the original PID to the user. Otherwise, a new PID is assigned. If computation of the SHA-1 checksum is expensive, we may chose to determine the last update timestamp of any data item addressed by the query (including deleted items). If none of these timestamps is later than the last query execution timestamp of the set of identical queries, then no new PID needs to be assigned as result sets will be identical.

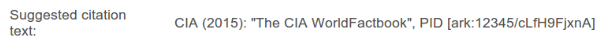
### 3.3.6 R9 - Store the Query

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

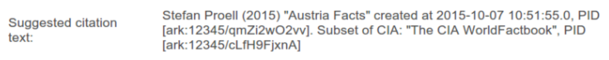
The query store facilities presented in *R3* provide the technical infrastructure for storing the query details and associated metadata. Each query is uniquely identified by its PID.

In order to avoid updates of the data during the process of storing the query in the query store, the system needs to implement a transaction concept, which locks the table during the subset creation process and ensures atomicity (A). In multi-user environments, we need to ensure that a second query executed simultaneously does not alter the data that the first user is currently processing, as this would result in a different subset. For this reason, we apply the isolation (I) principle, which prevents concurrent modifying access. A transaction in our scenario consists of the following steps:

1. Ensure stable sorting and normalise query (*R5*).
2. Compute query hash (*R4*).
3. Open transaction and lock tables (I)
4. Execute (original) query and retrieve subset.
5. Assign the last global update timestamp to the query (*R7*).
6. Close transaction and unlock tables (A)
7. Compute result set verification hash (*R6*).
8. Decision process:
   (a) Decide if the query requires a new PID (*R8*). If so:
   (b) Persist metadata and query (*R9*).


Suggested citation text: CIA (2015): "The CIA WorldFactbook", PID [ark:12345/cLfH9FjxnA]

(a) Entire Data Set


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]

(b) Subset of the Data Set

Figure 2: Automatically Generated Citation Texts

#### *3.3.7 R10 - Automated Citation Text Generation*

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

Researchers publish their findings usually in journals and conference proceedings. Depending on the designated community, several citation styles and principles exist. Many disciplines either utilise a specified citation format or they write their papers with LaTeX and include the references to the related work with the BibTeX citation system. To lower the barrier for researchers, the query store should produce such citation snippets, which can be easily integrated into publications with a copy and paste approach.

The query store contains metadata about a data set. In addition, the query store may also contain further metadata fields, such as a title for the data set and a specific subtitle for the subset. Users can be prompted for specifying this information during the ingest phase of a data set and during the subset creation process. Both - the entire data set and a subset - are identifiable by a unique PID, which is also stored in the query store. The citation texts must at least contain the author, the creation date and the PID of a data set. Researchers can then use this automatically generated citation text in an appropriate format, for instance BibTeX, and include this snippet directly into their publications and reports. This lowers the burden of citing data sets and allows to identify and retrieve a subset at a later point of time.

*Example*: In order to allow researcher to create citable subsets of CSV data, an interface may allow users to upload individual CSV files into the system. During the upload process, they are asked to provide a title for the data set, an author name and a description text. Upon the creation of a subset, the user also provides a specific subset title and a short description. Figure 2a shows the citation text snippet of a CSV file, including its PID and the year.

Figure 2b shows the citation text snippet of a subset, created from the parent data set described in 2a. It references the PID of the parent data set and the subset and includes a fine granular timestamp for the subset. Note that this example adopts a citation style for subsets from datasets analogous to papers in proceedings, listing the creator of the query (the persons intellectually responsible for putting together the specific subset, analogous to the author of a paper), as well as the persons responsible for maintaining the entire dataset (analogous to the editor of a proceedings volume). Both snippets are provided at the landing page (see *R11*) and can be copied from the browser into a publication.

This example demonstrates the generation of citation snippets for standard flat data sets. In hierarchically structured data with different contributors and differing citation requirements for various levels or branches of the hierarchy, where the information composing the citation is scattered across different elements of the dataset, more complex approaches for generating citation snippets are required, as e.g. presented in [4, 6].

## 3.4 Resolving PIDs and Retrieving the Data

The following recommendations deal with the retrieval of the data and metadata.

#### *3.4.1 R11 - Landing Pages*

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

Data sets and subsets are uniquely identifiable by their PID, which resolves to a human readable landing page. Typically, a landing page is reachable by a unique URL and presented in a Web browser. The landing page presents the metadata of a data set in a human readable way and provides links to other versions of the same data set, related data sets and to the data as download.

Landing pages are essential for describing data sets for the long term. As their format is usually simple and their content can easily be archived with standard Web crawling techniques, they can be preserved for the long term, even if the actual data may not exist anymore. Providing such tombstone landing pages is necessary for keeping the evidence of existence of a data set, even beyond the life time of the data.

Upon activating a download link (or other means of an access request, e.g. passing the data to an analysis API), the query is re-executed. Per default, the query is re-executed with the timestamp of the original query execution. However, in principle, the timestamp can be chosen freely. For retrieving the data from a provided PID, the system can allow to specify a date as parameter, thus allowing to retrieve semantically equivalent subsets from different points in time. This allows to measure the evolution of specific subsets and retrieving the respective subset with e.g. all additions and corrections applied up until the most recent state of the data store.

There might be situations when data might not be available anymore. Although long-term persistence is a key requirement for data repositories, technological and especially organisational change may prevent the data provider from keeping the data. Legal requirements are also a reason, why a data set might not be available any more. Deleting single records due to legal obligations may prevent subsets from being reproducible, hence they cannot be retrieved any more. These effects need to be reflected on the landing page, i.e. the user needs to be made aware that the data set may not be available for download anymore. Yet the metadata at the landing page still can provide useful information, such as a reference of the closest fit of available subsets or references to earlier or later views on the data sets.

*Example*: Typical landing pages provide the information stored in the query store. This includes creators, creation date, a description and additional information such as parent- and sibling data sets, number of records, hash codes etc. The landing page also provides a download button, which triggers the automatic re-execution of the query and the download of the data set, if access permissions are granted. Note that, for the user, the fact that the PID resolves to a query that gets re-executed rather than an actual subset, is entirely transparent. Whether the actual query string (and normalised/timestamped query) is presented to the user will depend on the community. In its basic setting, for the user the PID directly identifies the subset. Whether this exists directly as a file or is dynamically created is not relevant.

### *3.4.2 R12 - Machine Actionability*

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

Landing pages presented in Web browsers enable human users to read and interpret the metadata of a subset. As experiments are increasingly executed by workflow engines, which produce and consume data in an automated fashion, there is a need to provide machine actionable landing pages as well. For this reason a programmatic approach is needed, which allows programs and machines to resolve PIDs, read and interpret the metadata and retrieve the appropriate data sets without human interaction. This allows the PID to be used directly as input parameter in a data driven process, allowing the automatic retrieval and provisioning of the data to the process.

In order to provide a stable API for data identification and data citation, the interfaces, methods and parameters need to be specified and documented.

*Example*: Metadata can be provided in structured documents, such as CSV, XML or JSON. All three formats provide schemata for defining the fields and their data types. This is essential of the data is processed automatically. The API may also provide access to the data in different formats based on their availability.

## 3.5 Upon Modifications to the Data Infrastructure

Technological advancement is a major reason for change in information systems. The following recommendations describe how the resulting changes should be accommodated within the data infrastructure to ensure sustainable resolution of the dynamically generated subsets.

### *3.5.1 R13 - Technology Migration*

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

As technology evolves, data may be shifted to new environments (new database technologies, adapting the data representation, etc.). As subsets are dynamically created we need to ensure, that the queries identifying the subsets will also work in the new settings. Thus, together with the migration of the data to a new environment, the queries need to be migrated accordingly. As with all kinds of transformations, the existing queries should remain stored as part of the provenance trail, adding the re-written query adapted to the new data store technology. If the data representation should change in such a way that no identical hash input transformation can be created, new fixity information (hash key/checksum) should be calculated and added to the metadata. The PID resolution may need to be redirected if the location of the landing page changes.

Simple technology updates (such as a new database version) usually do not have effects on the results. Nevertheless, the subsets need to be verified after each upgrade. Complex migrations can be cumbersome and require comprehensive checks for their effects on the overall infrastructure (schemata, API, services etc). If the query language is affected by the migration of the system or a completely different database system is used, all the queries need to be migrated (i.e. translated) into the new system as well. It needs to be ensured, that any new query language is equally powerful and produces the same results.

*Example*: One of our first prototypes for CSV files used Git in order to store versioned copies of the data. For obtaining subsets, we used a CSV2JDBC driver and simple SQL statements to specify which records to include. The query store stored the parameters (selected columns, applied filters, used sorting) and linked to the revision of the CSV file in the Git repository. In order to improve the performance of the prototype solution, we migrated the CSV files into a relational database scheme, by parsing them and creating the appropriate tables on the fly. Each CSV file was then imported into a separate table, where the file name served as the table name. Versioning now was done by the database system, by inserting a new updated record, a timestamp of the event and the event type for each update. We could reuse the queries stored in the query store, by re-writing the queries to support the versioning scheme of the database system. As we had the parameters available and knew the file name, the queries could be mapped to the new system. As the RDBMS supports indices, data access times could be greatly improved.

### *3.5.2 R14 - Migration Verification*

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

After any changes to the system, it needs to be ensured that all subsets can be retrieved and that their content is identical. This can be achieved via the fixity information (hash keys/checksums) stored in the query store (R3) as computed originally when the data subset was generated. It needs to be ensured, that the hash input representation, i.e. the representation of the data over which the fixity information is computed, is identical.

Some migration paths might not allow or, at least, make it unnecessarily complex to directly compare the records and their hash keys respectively, due to different formats or the unavailability of suitable hash functions or encodings of the information (e.g. a switch from ASCII representation of data to Unicode). In this case, all the records need to be compared pairwise in the old and the new system, to make sure that all records are still contained in both systems. Another possibility is to provide a verification tool, which wraps the functionality of the old system and allows comparing the records in the new environment. Subsequently, new fixity information may be computed over the new data representation and added to the metadata stored in the query store. For our simple CSV prototype, we computed an SHA-1 hash of each CSV file and of each exported CSV subset, by using the Linux command line tool md5sum. This hash key was stored in the query store. The database system that we used in the improved version does not allow to compute such SHA-1 hashes directly, as the structure of the table was changed. For this reason, we implemented a wrapper tool, which allows exporting each subset as a CSV file. In a next step, we compute the hash key by using again md5sum and compared the result with the hash key stored in the query store. This process has to be done once for all subsets stored in the query store, in order to verify the correctness of the migration process.

## 4. CONCLUSIONS

In this work we presented 14 recommendations on how to make dynamic data identifiable. The query store (R3) is based upon persistent query identification (R8) and storage (R9), which allows identifying unique queries (R4) on versioned data (R1). Based on tracking and timestamping all changing events (R1, R2) and query timestamping (R7), a specific subset can be retrieved by re-executing the query. Stable sorting (R5) and verification metadata (R6) ensure that the significant properties of a subset remain intact. The query store provides additional features such as automatically generated citation texts of the subsets (R10) as well as human (R11) and machine readable (R12) landing pages. Technology migration (R13) and the verification of migration paths (R14) ensure long term compatibility and availability. Figure 3 summarises the recommendations as components of a system.

The proposed solution has several benefits compared to current approaches relying on individual data exports for each data set or ambiguous natural language descriptions of data set characteristics.

First of all, it allows identifying, retrieving, citing and sharing the precise data set with minimal storage overhead by only storing the versioned data and the queries used for creating the data set. In many environments data versioning is considered a best practice and often already in place, thus the implementation overhead is low.

Secondly, the approach allows retrieving the data both as it existed at a given point in time as well as the current view on

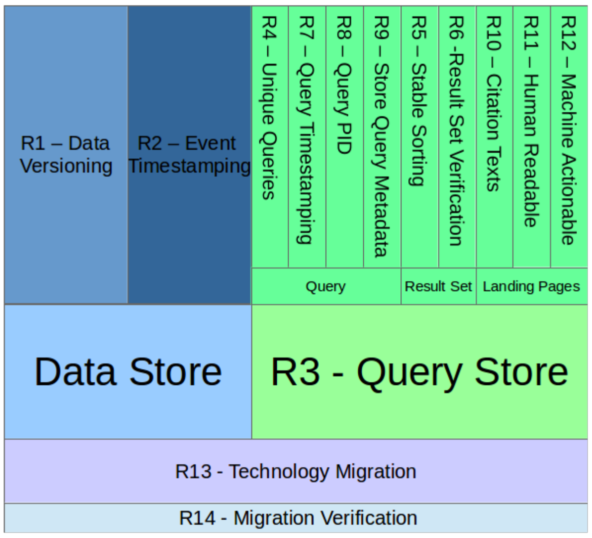


Figure 3: The Recommendations for Data Citation as Components

it, by re-executing the same query with the stored or current timestamp, thus benefiting from all corrections made since the query was originally issued. This allows tracing changes over time and comparing the effects on the result set. In addition the approach also allows to show verifiably and cite the fact, that a query has returned the empty set.

Thirdly, the query store as a basis for identifying the data set provides valuable provenance information on the way the specific data set was constructed, thus being semantically more explicit than a mere data export.

Last, but not least, the query store offers a valuable basis for understanding and analysing data usage, i.e. identifying, which parts of the data are being used. It also allows to identify reversely which subsets may be affected by corrections to the data and thus allows the creation of notification services to identify study results that may be impacted by such changes.

The recommendations are applicable across different types of data representation and data characteristics. This includes big or small data, static or highly dynamic and it is independent of the size of a data set. The same approach can be applied to identifying any form of subset, from the entire database down to an individual record or number, including even an empty result set. The solution is robust over time, as it supports the migration to new technologies. In order to improve trust, the (sub)sets can be verified for their completeness and correctness across technological boundaries using fixity information. In terms of required investments in infrastructure and resources, our solution requires the implementation and development of data versioning and hashing, the development of suitable subsetting tools and the query store as well as the adaption of policies for the assignment of PIDs or the migration of data to different technology stacks.

These recommendations are currently being evaluated and implemented or deployed by a number of data centres and projects, including the Virtual Atomic and Molecular Data

Centre (VAMDC), the ENVRIplus project, i2b2, WUSTL, BCO-DMO, the Argo Buoy Network and others.

## 5. REFERENCES

[1] D. Agrawal, A. El Abbadi, S. Antony, and S. Das. Data management challenges in cloud computing infrastructures. In S. Kikuchi, S. Sachdeva, and S. Bhalla, editors, *Databases in Networked Information Systems*, volume 5999 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin Heidelberg, 2010.

[2] M. Altman. A fingerprint method for scientific data verification. In *Advances in Computer and Information Sciences and Engineering*, pages 311–316. Springer, 2008.

[3] M. Altman and M. Crosas. The evolution of data citation: From principles to implementation. *IASSIST Quarterly*, 37, 2013.

[4] P. Buneman, S. Davidson, and J. Frew. Why data citation is a computational problem. Preprint `http://frew.eri.ucsb.edu/private/preprints/bdf-cacm-data-citation.pdf` to appear in CACM, Retrieved at 29.03.2016, 2016.

[5] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan. Archiving scientific data. *ACM Trans. Database Syst.*, 29(1):2–42, March 2004.

[6] P. Buneman and G. Silvello. A rule-based citation system for structured and evolving datasets. *IEEE Data Eng. Bull.*, 33(3):33–41, 2010.

[7] 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.

[8] CODATA-ICSTI. Out of cite, out of mind: The current state of practice, policy, and technology for the citation of data, 2013. CODATA-ICSTI Task Group on Data Citation Standards and Practices.

[9] J. T. Dudley and A. J. Butte. Reproducible in silico research in the era of cloud computing. *Nature biotechnology*, 28(11):1181, 2010.

[10] R. E. Duerr, R. R. Downs, C. Tilmes, B. Barkstrom, W. C. Lenhardt, J. Glassy, L. E. Bermudez, and P. 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] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, Aug 1991.

[12] J. Freire, P. Bonnet, and D. Shasha. Computational reproducibility: state-of-the-art, challenges, and database research opportunities. In *Proceedings of the ACM SIGMOD International Conference on Management of Data. 2012*, pages 593–596. ACM, 2012.

[13] K. Hinsen. A data and code model for reproducible research and executable papers. *Procedia Computer Science*, 4:579 – 588, 2011. Proceedings of the International Conference on Computational Science (ICCS), 2011.

[14] C. S. Jensen and R. Snodgrass. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, 1999.

[15] K. Kulkarni and J.-E. Michels. Temporal features in sql:2011. *SIGMOD Rec.*, 41(3):34–43, oct 2012.

[16] J. Lin, M. Crane, A. Trotman, J. Callan, I. Chattopadhyaya, J. Foley, G. Ingersoll, C. Macdonald, and S. Vigna. Toward reproducible baselines: The open-source ir reproducibility challenge. In *Advances in Information Retrieval: 38th European Conference on IR Research. ECIR 2016.*, pages 408–420. Springer, 2016.

[17] R. Mayer and A. Rauber. A quantitative study on the re-executability of publicly shared scientific workflows. In *11th International Conference on e-Science (IEEE eScience 2015)*, 8 2015.

[18] J. P. Mesirov. Computer science. accessible reproducible research. *Science (New York, NY)*, 327(5964), 2010.

[19] S. Pröll 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.

[20] S. Pröll 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.

[21] S. Pröll 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.

[22] S. Pröll 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.

[23] 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/rda-wgdc-recommendations-vers-sep-24-2015.html, September 2015. Draft - Request for Comments.

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

[25] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys (CSUR)*, 31(2):158–221, 1999.

[26] G. Silvello. A methodology for citing linked open data subsets. *D-Lib Magazine*, 21(1):6, 2015.

[27] K. Torp, C. S. Jensen, and R. T. Snodgrass. Effective timestamping in databases. *The VLDB Journal*, 8(3-4):267–288, February 2000.

[28] J. Zhao, J. Gomez-Perez, K. Belhajjame, G. Klyne, E. Garcia-Cuesta, A. Garrido, K. Hettne, M. Roos, D. De Roure, and C. Goble. Why workflows break: Understanding and combating decay in taverna workflows. In *8th International Conference on eScience (IEEE eScience 2012)*, pages 1–9, Oct 2012.