Exploring the Evolution and Provenance of Git Versioned RDF Data

Natanael Arndt[a], Patrick Naumann[b] and Edgard Marx[a,b]

[a] Agile Knowledge Engineering and Semantic Web (AKSW)
Institute of Computer Science, Leipzig University
Augustusplatz 10, 04109 Leipzig, Germany
{arndt|marx}@informatik.uni-leipzig.de
[b] Hochschule für Technik, Wirtschaft und Kultur Leipzig (HTWK)
Gustav-Freytag-Str. 42A, 04277 Leipzig, Germany
patrick.naumann@stud.htwk-leipzig.de

Abstract The distributed character and the manifold possibilities for interchanging data on the Web lead to the problem of getting hold of the provenance of the data. Especially in the domain of digital humanities and when dealing with Linked Data in an enterprise context provenance information is needed to support the collaborative process of data management. We are proposing a possibility for capturing and exploring provenance information, based on the methodology of managing RDF data in a tool stack on top of the decentralized source code management system Git. This comprises a queriable history graph, the possibility to query arbitrary revisions of a Git versioned store and in the minimal granularity the possibility to annotate individual statements with their provenance information.

1 Introduction

Due to the distributed and collaborative character of Linked Data and the Web of Data in general tracking and exploring the provenance of data as well as versioning of the data, plays a key role. Information about the same resource can be accessible from different sources. Those information, may differ or contradict each other, especially in curatorial work of different parties in a specific context. Provenance may help to make assumptions about each source and help to choose an appropriated version of the data.

Furthermore, in collaborative processes, provenance provides a good basis for mechanisms to track down and debug the sources of errors. Tracking provenance while allowing changes to data is a basic requirement for any version control system. Therefore it is important to track the provenance of data at any step of a process involving possible changes of a dataset (e.g. creation, curation, linking).

The different scenarios with respect to a setup in the context of Linked Data can be described in the following three use cases. (1) Tracking provenance, when importing data sets from different sources on the Web of Data. (2) Record and explore the evolution of a vocabulary respective dataset during all update steps of a data management process. (3) Tracking down the source or point of introduction of a specific change to a dataset.

Especially in the domain of e-humanities when managing prosopographical data, such as the *Pfarrerbuch*[1] and the *Catalogus Professorum*[2] [16], it is crucial to track and be able to explore the provenance and evolution of the domain data. Further use cases are the *Heloise – European Network on Digital Academic History*[3] [15] and "Professorial Career Patterns of the Early Modern History"[4]. But also for business use cases as researched in the *LEDS – Linked Enterprise Data Services*[5] project and for managing library metadata as in the *AMSL*[6] [1,13] project, means for tracking down the origin of any statement introduced into a dataset in a collaborative data curation setup are needed.

Our aim for this work is to enable access to provenance-related metadata retrieved from RDF data which is managed in a Git repository. Describing provenance information requires a vocabulary that can be used to express the different aspects of the Git system. Further an appropriate transformation is required to access the metadata.

The paper is structured as follows. The state of the art and related work is presented and discussed in section 2. An analysis of the problem we are facing and requirements necessary for a solution are provided in section 3. We are presenting our approach in section 4 and demonstrate the approach using our prototypical implementation in section 5. Finally, a conclusion is given together with a prospect to future work in section 6.

Throughout the paper we are using the following RDF-prefix mappings: `prov`: `http://www.w3.org/ns/prov#`, `xsd`: `http://www.w3.org/2001/XMLSchema#`, `rdf`: `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, `rdfs`: `http://www.w3.org/2000/01/rdf-schema#`, `foaf`: `http://xmlns.com/foaf/0.1/`, `quitp`: `urn:tmp:quitp:`, `quit`: `urn:tmp:quit:`. While the URIs for `quit:` and `quitp:` are placeholders for the URIs produced by the *Quit Store* and an extension of the existing provenance vocabulary.

2 State of the Art and Related Work

In this section, we want to present currently available possibilities to express provenance as well as other metadata relevant to evolution tracking in RDF. With versioning systems in mind, we further summarize how metadata and archival information can be represented, stored and accessed. Finally, we provide a quick review of other projects that deal with the versioning of data. For our related work, we only consider approaches which provide or use provenance in a versioning context or can be used to support versioning systems.

[1] `http://aksw.org/Projects/Pfarrerbuch`
[2] `http://aksw.org/Projects/CatalogusProfessorum`
[3] `http://heloisenetwork.eu/`
[4] `http://catalogus-professorum.org/projects/pcp-on-web/`
[5] `http://www.leds-projekt.de/`
[6] `http://amsl.technology/`

2.1 Vocabularies

Currently, various vocabularies exist that allow the description of provenance. As a *World Wide Web Consortium* (W3C) Recommendation, the PROV ontology (PROV-O) [11] is the de-facto standard for the representation and exchange of domain-independent provenance. The *Open Provenance Model* [12] predates the PROV-O, but both use very similar approaches as their core components. Both vocabularies model provenance data as relations between *agents*, *entities* and *activities* or their respective equivalent.

Another popular standard for general-purpose metadata is *Dublin Core* respective the *Dublin Core Metadata Terms* (dct) [7]. The main difference to the prior ontologies is in their focus on expressing provenance. Both vocabularies provide means for expressing provenance metadata. While the PROV-O is more focused on activities that lead to a specific entity, Dublin Core focuses on the resulting entities.

The main advantage of using domain-independent vocabularies as a core is that they are usable by systems and tools that operate without any domain-specific knowledge. PROV-O-Viz[7] is an example of a visualization tool only working with the data expressed according to the PROV ontology.

2.2 Modelling Patterns

One way to annotate data is *reification* [6, "Reified Statements"]. In standard reification, a resource is used to denote a triple. The resource is attributed with the subject, predicate, and object of the triple it is identifying and referring to. Thus, metadata about the original triple can be supplied by attaching additional attributes to this reified resource. The downside of this approach is the increase of statements needed to describe the original triple, which is at least threefold. Additional effort is needed for querying and matching statements with their reified statement since every reified part has to be matched on its own.

Provenance can also be described using *named graphs*, whereby triples are extended with a fourth argument, the *context*. A context is denoted by a URI and thus can also be used as a subject for statements, allowing the description or annotation of graphs with provenance information. Temporal graphs [17] is a storage model utilizing named graphs to store the validity of triple sets, e. g. their temporal validity, allowing stateful queries for a given point in time, by searching all validity ranges containg that timestamp. Adepting this approach to a git-like environment, the temporal validity is best described by the commits a statement was added or removed. But their commit hashes cannot be used as range indicators, since hashes have no natural ordering preventing us from doing simple *between* checks.

As an alternative to the prior methods for storing metadata in RDF, *Singleton properties* [14] were introduced. Hereby, the predicates of a statement are replaced with a unique equivalent, e. g. by appending consecutive numbers.

[7] `http://provoviz.org/`

Those unique predicates, used as a resource, are attributed with their original predicates and other annotations, this allows to describe the original statement. This approach requires less overhead, in the number of triples, compared to reification. Using this concept, Queries against a data source have to be adjusted accordingly, since the original statements have changed.

2.3 Version Tracking Systems

Version control systems such as Subversion, Mercurial, and Git are common tools for versioning any kind of data but especially text-based formats. By using an appropriate serialization format, RDF data can also be stored in such versioning systems [3]. Any type of versioning system, since it is commonly used for collaborative work, provides some basic provenance information like a *committer*, *reasons for changes* and *timestamps* when changes happened. In the systems mentioned above, such information is stored in a non-semantic compliant manner thus can often only be accessed with tools provided by the respective version control system.

Git4Voc [10] describes a methodology on how to use the versioning system Git for collaborative vocabulary development. Further they describe, how to utilize the hook mechanism provided by Git to add additional functionality for *validation* and *documentation*. The *master* branch tracks different versions of the vocabulary while changes are done in other branches and merged back later. For *validation*, a combination of local and online tools is used. After the quality check is done and committed to the repository, a documentation for the vocabulary is generated. Git4Voc does not provide any semantically accessible provenance but only what is provided by Git itself. Existing tools (e.g. Git2PROV[8]) can provide a semantic compliant view using the PROV ontology.

The Git2PROV tool [5] allows to generate a provenance document using the PROV-Ontology for any public Git repository. It can be used as a web service or can be executed locally using Node.js[9]. Since our aim is to provide provenance for RDF data on graph- and triple-level Git2PROV isn't suited as a component since it is only able to handle provenance on a per-file-level.

R43ples [8] on the other hand is a version control system completely build on Semantic Web technology. Due to its design and internal structure versioning only happens on graph-level instead of instance-level, therefore versioning happens for each named graph but not for the whole dataset. Revisions are stored as deltas to its previous version to save storage space with the drawback of the extra time required to restore older revisions. Because of its Semantic Web only approach no command line tools like in Git are provided and all operations on the versioning system are done via the SPARQL interface. The interface therefore was extended with additional, non-standard keywords. It uses the RMO vocabulary, which is an extended and more domain-specific version of the PROV ontology.

[8] `http://git2prov.org/`
[9] `https://nodejs.org/`

Another approach is implemented by Stardog[10], a triple store with integrated version control capabilities. The versioning module provides functionality for tagging and difference generation between revisions. Revisions are on instance-level since a snapshot contains all named graphs from the time the snapshot was taken. RDF data and snapshots are stored in a relational database. The current state of the database can be queried via a SPARQL interface. For regular queries, the version history can be accessed by using the `SERVICE` keyword and the provided virtual service. The module can also execute queries on the version history metadata. While older states of the database can be restored, to our knowledge, they can't be queried directly. The provided provenance contains information about *time*, *committer*, an optional *message*, and *changesets* represented using a vocabulary based on PROV-O, extended by means to describe changesets.

The *Quit Stack* is meant to support collaboration on RDF datasets in distributed setups. The *Quit Store* [3], as a part of this tool stack, provides a framework for tracking and exchanging changes on a dataset in a distributed setup. Backed by a Git repository, it provides a quad store which holds the latest revision of all or selected RDF files. Even though any revision can be restored with a simple checkout, loading the data into the store requires a deserialization step beforehand. Through a SPARQL interface, each interaction with the store is recorded and in case of an update, the changed graphs are serialized and stored as a commit in the repository. The *Quit Store* does not provide any semantic form of provenance so far. Besides the *Quit Store*, the *Quit Stack* also provides a tool *Quit Diff* for comparing the graphs between recorded revisions [2]. The difference between the graphs can then be represented in various changeset vocabularies and as SPARQL Update query.

3 Problem Description

The size of provenance information, especially when providing evolutionary data of a resource, is likely to exceed the size of the data which it is describing. Therefore, we encourage the decoupling of data and provenance as an appropriate way for provenance access. As a conclusion, we state ways to access detached provenance information. For direct access, a provenance document available as a Linked Data resource can be provided. By providing a linking e.g. `prov:has_provenance` as HTTP-header or within the data itself, the provenance can be discovered in case of need. Provenance can be made available as a separate SPARQL service, which can be linked via `prov:has_query_service` and be used for federated queries.

The focus of our work is primarily meant as a methodology for handling provenance of Git versioned RDF data in general. The problem of selecting an appropriate level of provenance recording, also called *granularity*, depends on the use cases and technology of a specific system. To support developers of provenance systems, Groth et al. [9] provide general aspects of provenance

[10] `http://stardog.com/`

that should be considered by any provenance enabled system. The aspects were categorized as following:

Content describes what should be contained in provenance data, whereas entities, contributing sources, processes generating artifacts, versioning, justification, and entailment are relevant dimensions.

Management refers to concerns about how provenance should be captured and maintained, including publication and access, dissemination and how a system scales.

Use is about how user specific problems can be solved using recorded provenance. Mentioned are understanding, interoperability, comparison, accountability, trust, imperfections, and debugging.

While not all of these aspects are in our focus, they are meant to help on deciding on exact use cases and requirements. Based on those dimensions and the prerequisite of handling RDF data in a Git repository, we need to consider which processes have influenced the data and contributed to the data in form of commits. The aspect of versioning the data is covered by Git in combination with the *Quit Store*. For the category *management*, we have to evaluate how Git handles provenance and how internal provenance can be accessed by the user. As for *use*, we have already formulated three use cases in section 1. These use cases mainly cover, understanding, accountability, trust, and debugging. Interoperability is the main focus of the *Quit Store* and comparison the main focus of *Quit Diff*.

Furthermore the evolution of a dataset in a distributed setup is not necessarily happening in a linear manner. Multiple parties are creating different versions based on the same original data set (*branch*), while a consolidated version (*merge*) might be created at a later stage. This introduces the problem of finding the path, in which a change was introduced.

Resulting from our use cases and the mentioned categories we can formulate our requirements as follows:

1. A structured representation of metadata recorded to a specific version of a dataset
2. A queriable representation of the provenance information recorded for the evolution of a dataset
3. Random access to any version of the dataset
4. A possibility to analyze the origin of any individual statement in a dataset
5. The resulting system should be able to handle non-linear, i. e. branched and merged, revision histories.

4 Approach

In this section, we present our approach on how to extract and explore RDF versioned data in a distributed environment. It is build as a tool stack on top of Git for extending it with semantic capabilities. Our approach gains all of its versioning and storage capabilities from the underlying Git repository and the *Quit Stack* [3]. As the *Quit Framework* itself is planned to handle just RDF models without any further regards to semantics[3, Introduction], our approach follows this lead. Therefore, the two main concerns are, (1) to make the already existing provenance information from the version control system semantically available and (2) check how and to which extend additional and domain-specific metadata can be stored in the version control structure. We further introduce a methodology for fast access to different versions of a dataset using named graphs.

4.1 Storage and Data Structure

Non-semantical (Git)	Semantical (RDF)
Commit	`prov:Activity`
parent	`quitp:preceedingCommit`
author	`prov:wasAssociatedWith`
	`prov:qualifiedAssociation`
author date	`prov:startedAtTime`
committer	`prov:wasAssociatedWith`
	`prov:qualifiedAssociation`
committer date	`prov:endedAtTime`
message	`rdfs:comment`
Author and Committer	`prov:Agent`
name	`rdfs:label`
email	`foaf:mailbox`
Changeset	`quitp:updates`
	`quitp:graph`
	`quitp:addition/quitp:deletion`
Data	`prov:Entity`
	`prov:specializationOf`
	`prov:wasGeneratedBy`

Table 1. Semantic representation of a Git commit

Our initial effort was to transform the metadata, stored in the data model of Git to RDF making use of PROV-O. Table 1 provides an overview of attributes used to convert a Git commit. Commits in Git can be mapped to instances of the class `prov:Activity` associated with their author and committer. We follow the idea of De Nies et al. [5] and represent the start and end time of the Activity with the author and commit date of Git, which can be interpreted as the time till a change was accepted. PROV-O has no concept for commenting on activities, therefore we follow the suggestion of PROV-O and use `rdfs:comment` for commit messages. Git users are `prov:Agents`, stored with their provided name and email.

We represent Git names as `rdfs:label` since they do not necessarily contain a full name. For calculating changesets we use the strategy provided by *Quit Diff* [2]. Each update is linked to the original graph and described by graphs containing the added and deleted statements. We store every named graph contained in a file as an instance of `prov:Entity` bound to their respective object hash as it is used in Git. Further, we link it with the respective commit via `prov:wasGenerated By` and attribute its origin as `prov:specializationOf`. An excerpt concerning commits and entities is provided in listings 3 and 4 in section 5.

Our next step was the enrichment of the metadata in Git with additional information: e.g. the source specification of a dataset by recording its original URL on the Web or a SPARQL update query on a dataset resulting in a new commit. The main problem here was that Git itself offers no built-in feature for storing any user-defined metadata for commits or files. What Git offers instead is a functionality called `git notes`, which is like a commentary function on commits. Hereby, Git creates a private branch where text files, named after the commit they comment on, resides. While this may seem sufficient, notes can be edited like any other file under version control and our provenance would not be protected using Git's hashing and signing mechanism.

Thus, for extending our provenance information we have to dig a little deeper into Git's internal structures [4]. The internal storage structure of Git, basically, is just a file-system based key-value store. Git uses different types of objects to store both, structural information and data in files and addresses them with their sha1-hash[11]. The types used by Git to organize its data are *blobs*, *trees* and *commits*, the types are linked as shown in fig. 1.

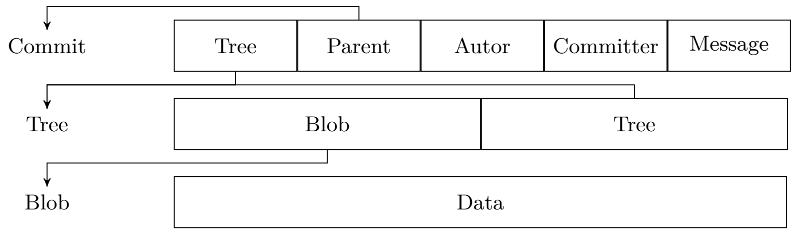

Figure 1. Internal structure used by Git

The content of any file that is put under version control is stored as *blobs* while folders are stored as *trees*. A *tree* consists of other *trees* and *blobs* respectively. Each revision is represented by a *commit*-object consisting of metadata, references to parent commits and a *tree*-object, which is considered as root. References such as *branches* and *tags* are simply files within Git, pointing to a commit as their entry point to the revision history.

This leaves us with only two options where additional metadata can be stored, *blobs* and *commits*. Even though every object type could be manually created

[11] Secure Hash Algorithm

and put into the Git storage, unused or unreferenced, objects might later be removed by Git's own garbage collection.

By using *blobs* we could store provenance data in the way as we store all our data, but we would have to split provenance in two different representational formats–semantical and non-semantical–which isn't desirable. Also, saving *blobs* means users are able to manually edit and commit them. While those changes can be reverted, having such a use case is also not desirable. Therefore our decision was to provide and obtain additional metadata as part of the commit message as shown in listing 1. Commit messages are unstructured data, meaning it won't break the commit when additional structured data is provided at the start or end of a message.

```
tree 31159f4524edf41e306c3c5148ed7734db1e777d
parent 3fe8fd20a44b1737e18872ba8a049641f52fb9ef
author pnaumann <patrick.naumann@stud.htwk-leipzig.de> 1487675007 +0100
committer pnaumann <patrick.naumann@stud.htwk-leipzig.de> 1487675007 +0100

Source: http://dbpedia.org/data/Leipzig.n3

Example Import
```

Listing 1. Git commit with additional data

More specific workflows can be added when needed. If the stored value contains line breaks, e.g. a query for a transformation is stored, we support a multi line syntax as shown in listing 2.

```
Query: "SELECT ?s ?p ?o {
        ...
        }"
```

Listing 2. Example for multi line key-value-pair in a commit message

The two workflows we currently support for *extract, transform, load* processes are *importing* and *transforming*. For an import we store `quitp:Import`, a subclass of `prov:Activity`, together with a `quitp:dataSource` property. A *transformation* is represented as a `quitp:Transformation` activity and a `quitp:query` property. Both are shown in table 2.

Non-semantical	Semantical
Import	`quitp:Import`
source	`quitp:dataSource`
Transformation	`quitp:Transformation`
query	`quitp:query`

Table 2. Semantic representation of metadata for describing contributions to an existing data set

4.2 Access to the Provenance Information

To access our provenance information we follow the recommendation of the "PROV-AQ: Provenance Access and Query" (W3C Working Group Note)[12]. We provide a SPARQL interface for each state in the history of the datasets in a Git repository as well as a SPARQL service for the provenance graph. The states and the provenance graph are built from the metadata provided by Git and combined with the additional metadata stored in the commit messages. To be able to query this information we have to transform it to RDF and store the resulting graph. This is done during a *synchronization* process. During synchronization, the store is built from the commits stored in Git, by traversing the Git commit history of every branch from its end, until a commit is found which already exists in the store. A quad store serves as cache, to decrease access time, by avoiding the complete parse procedure from the actual data stored in the Git repository. The depth of the synchronized history as well as a selection of the relevant branches is configurable according to the users' needs. Therefore the needed storage space can be reduced for devices with low storage capacities, at the cost of time for parsing graphs on-the-fly later on.

4.3 Random Access to Dataset Revisions

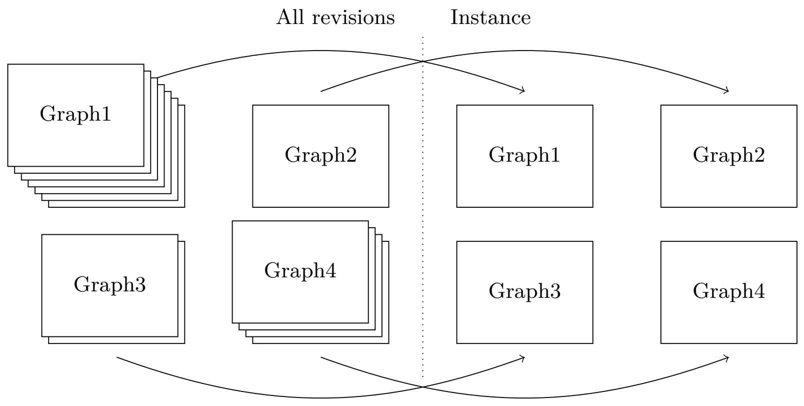

Figure 2. Creating a dataset from all available revisions of named graphs

The basic concept of Git is to reuse as many of its objects as possible for a new revision, whereby only snapshots of files that actually changed are created, instead of a full repository snapshot. Exploiting this storage structure, we are able to randomly checkout any Git commit in linear time. We create a revision of ever named graph in the repository, by using object hashes as a suffix. This

[12] `https://www.w3.org/TR/prov-aq/`

allows us to lookup all objects from Git's internal tree structure to create a virtual dataset, shown in fig. 2, containing the state of all graphs at that commit and run queries against it. Given a commit hash, we do this by rewriting the original URI of a named graph with their hashed equivalent of that commit and vice versa in query results.

5 Prototypical Demonstration

With the integration into the *Quit Stack* [3] in mind, we chose to build our prototypical implementation on the same technology, namely Python in combination with RDFlib[13] for handling the RDF files and SPARQL queries, Flask[14] for the HTTP API and pygit2[15] to interact with Git.

```
quitp:commit-f0c57d5b2b a prov:Activity, quitp:Import ;
    rdfs:comment """import: http://aksw.org/NormanRadtke

initial import""" ;
    prov:startedAtTime "2017-03-11T13:56:57+01:00"^^xsd:dateTime ;
    prov:endedAtTime "2017-03-11T13:56:57+01:00"^^xsd:dateTime ;
    prov:wasAssociatedWith quitp:user-e236a58c1c ;
    prov:qualifiedAssociation [ a prov:Association ;
            prov:agent quitp:user-e236a58c1c ;
            prov:role quitp:author, quitp:committer ] ;
    quitp:dataSource <http://aksw.org/NormanRadtke> ;
    quitp:updates quitp:update-46cadd74a5 .

quitp:commit-2cc30bc7f5 a prov:Activity, quitp:Import ;
    rdfs:comment """import: http://aksw.org/NatanaelArndt

imported natanael arndt""" ;
  prov:startedAtTime "2017-03-11T13:58:44+01:00"^^xsd:dateTime ;
    prov:endedAtTime "2017-03-11T13:58:44+01:00"^^xsd:dateTime ;
    prov:wasAssociatedWith quitp:user-e236a58c1c ;
    prov:qualifiedAssociation [ a prov:Association ;
            prov:agent quitp:user-e236a58c1c ;
            prov:role quitp:author, quitp:committer ] ;
    quitp:dataSource <http://aksw.org/NatanaelArndt> ;
    quitp:preceedingCommit quitp:commit-f0c57d5b2b ;
    quitp:updates quitp:update-c4e8149654 .

quitp:commit-d9fe8f514e a prov:Activity ;
    rdfs:comment "updated who knows who" ;
    prov:startedAtTime "2017-03-11T14:05:04+01:00"^^xsd:dateTime ;
    prov:endedAtTime "2017-03-11T14:05:04+01:00"^^xsd:dateTime ;
    prov:wasAssociatedWith quitp:user-71cd535876 ;
    prov:qualifiedAssociation [ a prov:Association ;
            prov:agent quitp:user-71cd535876 ;
            prov:role quitp:author, quitp:committer ] ;
    quitp:preceedingCommit quitp:commit-2cc30bc7f5 ;
    quitp:updates quitp:update-5bf3ae9594, quitp:update-cb82c36f12 .
```

Listing 3. Converted Git repository. We omitted parts and restrict this example on commits ...

Listings 3 and 4 show an example provenance graph created from a Git repository. For better illustration, we omitted parts of it, e.g. updates, agents,

[13] `https://rdflib.readthedocs.io/`
[14] `http://flask.pocoo.org/`
[15] `http://www.pygit2.org/`

and roles, for they are not required for this demonstration. The first part shows a full conversion of a repository, containing three commits and two named graphs, the second part shows how graphs were stored. Both graphs were copied from an external source, therefore a key-value pair was provided in the commit message (`rdfs:comment`) accordingly, as shown in listing 3. With this additional metadata a more specific activity, namely `quitp:Import`, was generated, together with the attribute `quitp:dataSource` for the external source.

```
quit:radtke-46cadd74a5 prov:specializationOf quit:radtke ;
    prov:wasGeneratedBy quitp:commit-2cc30bc7f5 .

quit:arndt-c4e8149654 prov:specializationOf quit:arndt ;
    prov:wasGeneratedBy quitp:commit-2cc30bc7f5 .

quit:arndt-cb82c36f12 prov:specializationOf quit:arndt ;
    prov:wasGeneratedBy quitp:commit-d9fe8f514e .

quit:radtke-5bf3ae9594 prov:specializationOf quit:radtke ;
    prov:wasGeneratedBy quitp:commit-d9fe8f514e .
```

Listing 4. ... and graph revisions

As listing 4 shows, four blobs were stored by Git in those three commits. One blob for each import, since Git needed to create a file holding the imported named graph. The other two commits were created upon the third commit, where both graphs were edited. This example shows, that the second commit only required a partial snapshot, without losing any information by reusing blobs from the previous commit. Next, we will explain how to obtain a graph containing the exact state from that commit. When a state is requested by its commit, we generate a mapping for all named graphs that existed in the given commit. Because of the additional overhead, not all object-to-commit mappings exist in the provenance graph, only those which create an object. If an object is unchanged for a commit, the mapping is omitted, but can be taken from Git's internal data structure.

```
100644 blob c4e8149654    arndt.nt
100644 blob 46cadd74a5    radtke.nt
```

Listing 5. Content of the *tree* object from the second commit. Showing two contained files and their hashes

```
{ 'quit:radtke': 'quit:radtke-46cadd74a5',
  'quit:arndt': 'quit:arndt-c4e8149654' }
```

Listing 6. Mapping for instance graph based on Git hashes

For our example, listing 5 shows how Git stored the two named graphs from the second commit in a *tree* object, while listing 6 shows our mapping, which allows us to access any graph with the state of the second commit. As shown in the listings above, we can obtain the missing hashes for our mapping from Git. Any query to a graph including such mapping is rewritten to match our stored partial snapshots instead of the requested named graphs, and translated back after the request has finished.

5.1 Quit Blame

As an example for the usage of provenance, similar to the functionality of `git blame`, we have also built a method to retrieve the origin of each individual statement in a dataset and associate it with its entry in the provenance graph. Given an initial commit, we traverse the Git history to find the actual commit of each statement when it was inserted and annotate it with the metadata for that commit.

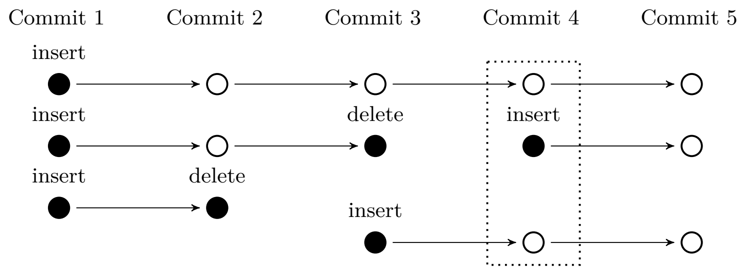

Figure 3. Example for an insert/delete chain in Git used by `git-blame`

The three statements that exist in the fourth commit of fig. 3 therefore, should be matched with the commits 1, 4 and 3 respectively, since those are the commits where the statements were introduced. We are utilizing SPARQL 1.1 features to provide values to a query, we list all quads without an annotation and query our provenance graph with the query shown in listing 7. We also bind the *?commit* variable to the given commit.

```
SELECT ?s ?p ?o ?context ?commit ?name ?date WHERE {
  ?commit prov:endedAtTime ?date ;
          prov:wasAssociatedWith ?user ;
          quitp:updates ?update .
  ?user   foaf:mbox ?email ;
          rdfs:label ?name .
  ?update quitp:graph ?context ;
          quitp:additions ?additions .
  GRAPH ?additions {
    ?s ?p ?o
  }
  VALUES (?s ?p ?o ?context) {
    ...
  }
}
```

Listing 7. Query for `git blame` implementation

Using this method on the second commit of the example from the previous section, we get the results presented in table 3:

6 Conclusion

Tracking and exploring provenance is crucial in a collaborative environment, as the Web of Data. In this paper we have examined, how metadata and datasets stored in a Git repository can be enriched, processed and used semantically. We've built this work on top of the foundation provided by the *Quit Stack* and

subject	predicate	object	context	commit	committer	timestamp
aksw:NormanRadtke	foaf:mbox	mailto:radtke@...	quit:radtke	f0c57d5b2b	pna	2017-03-11T13:56:57+01:00
aksw:NormanRadtke	rdfs:type	foaf:Person	quit:radtke	f0c57d5b2b	pna	2017-03-11T13:56:57+01:00
aksw:NatanaelArndt	foaf:mbox	mailto:arndt@...	quit:arndt	2cc30bc7f5	pna	2017-03-11T13:58:44+01:00
aksw:NatanaelArndt	rdfs:type	foaf:Person	quit:arndt	2cc30bc7f5	pna	2017-03-11T13:58:44+01:00

Table 3. Result of `git blame` adaption for the second commit

added methodologies for how Git commits, their metadata and datasets can be used for provenance. For functionality, we adapted the `git blame` command to `quit blame` for semantic data. Thus we are able to track the provenance on any update operation in the dataset (use cases 1 and 2, section 1; requirement 1 section 3). With the provenance graph, we are also able to explore the recorded data using SPARQL (use case 2, requirement 2) and due to its graph structure we are also able to represent any kind of a branched and merged history (requirement 5). Using `quit blame` we are able to track down the origin of any individual statement in a dataset (use case 3, requirement 4). With our random access query interface we can also execute SPARQL queries on the store at the status of any revision (requirement 3).

With the presented system we can provide access to the automatically tracked provenance information with semantic web technology in a distributed collaborative environment. In future research, one of the biggest challenges might be, storing the RDF metadata in a more efficient way. Further the query performance and storage overhead of the current prototypical implementation has to be investigated.

7 Acknowledgements

This work was partly supported by a grant from the German Federal Ministry of Education and Research (BMBF) for the LEDS Project under grant agreement No 03WKCG11C and the DFG project *Professorial Career Patterns of the Early Modern History: Development of a scientific method for research on online available and distributed research databases of academic history* under the grant agreement No GL 225/9-1.

References

1. Arndt, N., Nuck, S., Nareike, A., Radtke, N., Seige, L., Riechert, T.: AMSL: Creating a linked data infrastructure for managing electronic resources in libraries. In: Horridge, M., Rospocher, M., van Ossenbruggen, J. (eds.) Proceedings of the ISWC 2014 Posters & Demonstrations Track. CEUR Workshop Proceedings, vol. Vol-1272, pp. 309–312. Riva del Garda, Italy (Oct 2014)
2. Arndt, N., Radtke, N.: Quit diff: Calculating the delta between rdf datasets under version control. In: 12th International Conference on Semantic Systems Proceedings (SEMANTiCS 2016). SEMANTiCS '16, Leipzig, Germany (Sep 2016)

3. Arndt, N., Radtke, N., Martin, M.: Distributed collaboration on rdf datasets using git: Towards the quit store. In: 12th International Conference on Semantic Systems Proceedings (SEMANTiCS 2016). SEMANTiCS '16, Leipzig, Germany (Sep 2016)
4. Chacon, S., Straub, B.: Pro git. Apress (2014)
5. De Nies, T., Magliacane, S., Verborgh, R., Coppens, S., Groth, P., Mannens, E., Van de Walle, R.: Git2prov: exposing version control system content as w3c prov. In: Proceedings of the 2013th International Conference on Posters & Demonstrations Track-Volume 1035. pp. 125–128 (2013)
6. Dodds, L., Davis, I.: Linked data patterns (May 2012), `http://patterns.dataincubator.org/`
7. Dublin Core Metadata Initiative, et al.: DCMI metadata terms (2004), `http://dublincore.org/documents/dcmi-terms/`
8. Graube, M., Hensel, S., Urbas, L.: R43ples: Revisions for triples. In: Proceedings of the 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems (SEMANTiCS 2014) (2014)
9. Groth, P., Gil, Y., Cheney, J., Miles, S.: Requirements for provenance on the web. International Journal of Digital Curation 7(1), 39–56 (2012)
10. Halilaj, L., Grangel-González, I., Coskun, G., Auer, S.: Git4voc: Git-based versioning for collaborative vocabulary development. In: 10th International Conference on Semantic Computing. pp. 285–292. Laguna Hills, California (Feb 2016)
11. Lebo, T., Sahoo, S., McGuinness, D., Belhajjame, K., Cheney, J., Corsar, D., Garijo, D., Soiland-Reyes, S., Zednik, S., Zhao, J.: Prov-o: The prov ontology. W3C recommendation 30 (2013)
12. Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., et al.: The open provenance model core specification (v1. 1). Future generation computer systems 27(6), 743–756 (2011)
13. Nareike, A., Arndt, N., Radtke, N., Nuck, S., Seige, L., Riechert, T.: AMSL: Managing electronic resources for libraries based on semantic web. In: Plödereder, E., Grunske, L., Schneider, E., Ull, D. (eds.) Proceedings of the INFORMATIK 2014: Big Data – Komplexität meistern. GI-Edition—Lecture Notes in Informatics, vol. P-232, pp. 1017–1026. Gesellschaft für Informatik e.V. (Sep 2014), © 2014 Gesellschaft für Informatik
14. Nguyen, V., Bodenreider, O., Sheth, A.: Don't like rdf reification?: making statements about statements using singleton property. In: Proceedings of the 23rd international conference on World wide web. pp. 759–770. ACM (2014)
15. Riechert, T., Beretta, F.: Collaborative research on academic history using linked open data: A proposal for the heloise common research model. CIAN-Revista de Historia de las Universidades 19(0) (2016)
16. Riechert, T., Morgenstern, U., Auer, S., Tramp, S., Martin, M.: Knowledge engineering for historians on the example of the catalogus professorum lipsiensis. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) Proceedings of the 9th International Semantic Web Conference (ISWC2010). Lecture Notes in Computer Science, vol. 6497, pp. 225–240. Springer, Shanghai, China (2010)
17. Tappolet, J., Bernstein, A.: Applied temporal rdf: Efficient temporal querying of rdf data with sparql. In: European Semantic Web Conference. pp. 308–322. Springer (2009)