Distributed Collaboration on RDF Datasets Using Git

Towards the Quit Store

Natanael Arndt
Universität Leipzig
Augustusplatz 10
04109 Leipzig, Germany
arndt@informatik.uni-leipzig.de

Norman Radtke
Universität Leipzig
Augustusplatz 10
04109 Leipzig, Germany
radtke@informatik.uni-leipzig.de

Michael Martin
Universität Leipzig
Augustusplatz 10
04109 Leipzig, Germany
martin@informatik.uni-leipzig.de

ABSTRACT

Collaboration is one of the most important topics regarding the evolution of the World Wide Web and thus also for the Web of Data. In scenarios of distributed collaboration on datasets it is necessary to provide support for multiple different versions of datasets to exist simultaneously, while also providing support for merging diverged datasets. In this paper we present an approach that uses SPARQL 1.1 in combination with the version control system Git, that creates commits for all changes applied to an RDF dataset containing multiple named graphs. Further the operations provided by Git are used to distribute the commits among collaborators and merge diverged versions of the dataset. We show the advantages of (public) Git repositories for RDF datasets and how this represents a way to collaborate on RDF data and consume it. With SPARQL 1.1 and Git in combination, users are given several opportunities to participate in the evolution of RDF data.

CCS Concepts

• **Information systems** → **Data management systems; Network data models; Distributed database transactions; Data federation tools; Version management;** *Resource Description Framework (RDF);* • **Software and its engineering** → Collaboration in software development;

Keywords

co-evolution, distributed version control system, distributed collaboration, git, SPARQL Update, rdf dataset

1. INTRODUCTION

Collaboration of people and machines is a major aspect of the World Wide Web and especially on the Semantic Web. Currently the access to RDF data on the Semantic Web is possible following the SPARQL specification [18] and the Linked Data principles [13]. This allows clients to access and retrieve data stored and published by central services and databases.

The collaboration on such Linked Data Sets, currently is mainly done by keeping a central version of a dataset, and collaborators are editing on the same instance simultaneously. This situation has drawbacks in different scenarios, where multiple different versions of the dataset should or could exist simultaneously. The existence of multiple different versions of a dataset occurs, if not all participants simultaneously have access to the same database management system, for instance if they are working from mobile devices with limited network connection. Further the establishment of multiple different versions might be necessary, even on a common database management system. For instance, when in an ongoing discussion a consensus on a certain topic is not yet reached, or different releases of a dataset should be available.

In software development the same problem exists, when a distributed team is simultaneously working on a common code basis. Especially if one version of a program includes new features, while a different version should consist only of well tested functionality. A common methodology to address this issue is, that every developer has her own copy of the program source code and can independently implement new features or fix bugs, while one or more common synchronized versions of the source code are created using a version control system capable of dealing with concurrent versions.

Transfered back to the Linked Open Data Cloud the subject of collaboration are datasets resp. graphs, instead of the source code files for a software program. Collaborators are individual data scientists and domain experts curating local versions of a dataset instead of programmers. For the Semantic Web now the overall question is: How can the collaborative curation of distributed Linked Data Knowledge Bases be synchronized?

Our aim is to provide a system that enables distributed collaboration of domain experts and data scientists on RDF datasets. For this purpose we are proposing a methodology of using a SPARQL 1.1 interface in combination with Git and we introduce the *Quit Store* ("Quads in Git") as abstraction layer between the repository and applications working with the RDF dataset. In contrast to many other approaches (cf. section 6) we are concentrating on a pure RDF data model, containing statements and don't look at additional semantics, like OWL or SKOS. This especially means we don't have to care about semantic conflicts, but

SEMANTiCS 2016, September 12–15, 2016, Leipzig, Germany

 ISBN 978-1-4503-4752-5/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2993318.2993328

only structural conflicts. We rather want to provide a solid foundation usable for collaboration on RDF data. Checks regarding a special semantic or even domain and application specific checks can be integrated on top of the Quit Store in custom merge tools or using quality assessment tools as e.g. proposed in [11]. Currently we are not aiming at big datasets (gigabytes or millions of triples), we rather want to investigate the methodology on datasets which can be curated by individual humans.

Within the Linked Enterprise Dataservices (LEDS)[1] project the topic of co-evolution for the management of background knowledge requires a system for synchronizing and distributed knowledge bases, as well as benchmarking relevant systems. Further use cases and requirements where formulated during the work on a history data project *Pfarrerbuch*[2] and the Héloïse platform[3] [15]. For the design of our system, we have especially concentrated on the following three use cases:

Collaboration, Synchronization and Exchange.

As stated before collaboration is one of the most important topics regarding the evolution of the World Wide Web and knowledge bases. Wiki systems have already provided a centralized technology to support people in collaborating in data creation. Using a distributed system can even loosen collaborators from the need of central platforms and from the need of working with the same user interface. This methodology of branches for public contributions should also be integrated with the Structured Feedback protocol [1].

Synthetic Dataset Dreation for Benchmarks and Tests.

Searching for benchmarking and test datasets in RDF for distributed co-evolution and collaboration systems only brought up the DBpedia Live changesets[4]. But this dataset contains a linear history and doesn't include branching and merging. Following the "chicken or the egg" problem it is hard to create such datasets without an existing platform. The Quit Store system should help to generate test data to evaluate different merge strategies on real RDF data and more complex co-evolution scenarios.

Backup.

When working with RDF data it is, as for other data, always important to create backups of the current work. Providing a specific tool that supports the data creator in tracking the changes of the data and synchronizing the data with a secure location, e.g. for open data a publicly hosted repository, helps to avoid additional superfluous steps in the daily workflow. Further a version controlled backup system can help to restore data even after faulty changes.

The paper is structured as follows: general requirements for a distributed collaboration and versioning setup are formulated in section 2, followed by relevant preliminaries, such as Git, thoughts about RDF serialization and Diff and blank nodes, in section 3. The methodology of the system is specified in detail in section 4. Further we are documenting our prototypical reference implementation in section 5. We are discussing the state of the art and related work in section 6. Finally a conclusion and an outlook on future work is given in section 7.

[1]http://www.leds-projekt.de/
[2]http://aksw.org/Projects/Pfarrerbuch
[3]Héloïse - European Network on Digital Academic History: http://heloisenetwork.eu/platform
[4]http://live.dbpedia.org/changesets/

2. REQUIREMENTS

In the following we present requirements relevant for a distributed collaboration system on RDF Datasets under consideration of co-evolution by using a distributed version control system. The requirements are structured in requirements coming from collaboration aspects and the versioning requirements.

2.1 Collaboration Requirements

Requirements for collaborative vocabulary development are already formulated in [9]. We are adopting the requirements formulated there, which are overlap with our requirements. But in contrast to [9] we mainly concentrate on the technical collaboration on a distributed network, rather than focusing on a specific use case, such as vocabulary creation.

Provenance of Contributions.

Provenance information should be attached to a contribution to the common dataset. This shall at least be a change reason, author information and date of commit. For the automatic interaction with the system, information which require manual interaction might be omitted. (cf. "Communication support (R1)" and "Provenance of information (R2)" in [9])

Roles and Access Rights.

The system should respect and be able to support different access conditions and restrictions for collaborating agents in different roles. The system should not introduce new roles and access conditions, other than used in the underlaying collaboration platform (this would also support but not presume "Different roles (R3)" as required in [9]).

Syntactical Robustness.

The collaborating platform should ensure, that a syntactically correct input RDF dataset on commit will result in a syntactically correct dataset again. The system should report, but not break on syntactical errors.

Support of RDF Datasets and Modularization of Graphs.

The system should be able to handle multiple RDF graphs, i.e. RDF dataset, in a repository. This allows users resp. collaborators to organize the stored knowledge in individual organizational units, as it is required by their application. This requirement also provides the functionality to implement the requirement "Modularity (R9)" as formulated in [9]. The method should work with different granularities of modularization of RDF datasets.

Heterogenous Repositories.

A repository can contain RDF Data alongside other files. This should allow usage scenarios, where RDF graphs are embedded in bigger projects, such as source code repositories.

Heterogenous Setup of Editors.
Different implementations of collaboration interfaces can access and collaborate on a common repository. Collaborators can use different RDF editors to contribute to the repository. To some extend the methodology should even be robust to manual editing of RDF files contained in the repository. In contrast to the requirement „Editor agnostic (R8)“ as formulated in [9], we don't require the syntax independence on the repository and understand the editor agnosticism as transparency of the interface.

2.2 Versioning Requirements

In the following we are formulating requirements which are mainly implied by the usage of a co-evolution approach for supporting the distributed collaboration.

Declarative Version Log.
The version log should contain declarative entries for a version of the RDF dataset, which can be used to retrieve any version in the history of the repository. Additionally to the declarative entries, imperative or procedural annotations can be added to help to reconstruct the semantics of the version log.

Deltas Among Versions.
It should be possible to calculate the substantial difference between versions generated by contribution of collaborators. The calculated difference should be expressed in a machine readable format. (cf. "Deltas among versions (R7)" in [9])

Support Version Log Operations.
The system should support various operations to work on a version log. The required operations are *merge*, *revert* (resp. *backout*), and *commit*. Further operations, such as *commute*, *rebase* and other operations for history editing might be helpful but are not directly required.

3. PRELIMINARIES

In the following we are giving a brief overview and introduction to technologies and design considerations, which are relevant for our methodology. For the basic technologies, we are giving a number of references for further reading and a detailed understanding. Design considerations which are serving as foundation for our methodology are briefly discussed and possible alternative decisions are pointed out.

3.1 Git

Git is a distributed version control system[5] designed to be used in software development. It is used for over 35 million projects on github[6] and can also be used on other platforms, such as bitbucket or gitlab and can be hosted on self controlled servers or used in a pear to pear manner as well. Git is very flexible in providing branching and merging strategies and synchronizing with multiple remote repositories. Due to the flexibility, best practices and workflows have been developed to support software engineering teams with organizing different versions of a programs source code, such as e.g. *gitflow*[7] and the *Forking Workflow*[8].

In contrast to other *version control systems* (VCS) such as Subversion or CVS[9], Git is a *distributed version control system* (DVCS), similar to Mercurial. As such, in Git, users work on a local version of a remote Git repository, which is a complete clone of the remote repository. Git operations, such as `git commit` are executed on the local system. The repository contains commits, which represent a certain version of the working directory. Each version of the working directory contains the current state of its files at the given version. Even so Git is mainly intended to work with text files, it is also capable of dealing with other binary files. Files are stored as a binary large object (blob), while equal files are stored as pointers to the corresponding blob of each file.

Out of the box, Git already provides the capability to store provenance information alongside a commit (cf. "Provenance of Contributions"). "Heterogenous Repositories" are also possible as Git doesn't put any limitations on the file types under version control. The Git version log can be considered as "Declarative Version Log", since each version exactly contains the status if the files as it is at the point of the commit creation. Using `git diff` it is also possible to get the "Deltas Among Versions" and Git supports *merge*, *revert*, *commit* as well as (interactive) *rebase* operations.

3.2 Serialization of RDF Data

RDF 1.1 specifies multiple different formats which can be used for serializing RDF graphs (RDF/XML[10], Turtle[11], RDFa[12], N-Triples[13]) and RDF datasets (TriG[14], JSON-LD[15], N-Quads[16]). RDF graphs and RDF datasets can be serialized in different formats and thus the same RDF statements can result in completely different textual representations and the resulting file size can vary. Even the same graph or dataset serialized twice in the same serialization format can be textually different. To allow a better readability and processability of the differences between two versions in the version control system (cf. section 2 "Deltas Among Versions"), we have to find an easy to compare default serialization format. For our approach we have decided to use the N-Quads serialization [4] in Git repositories. N-Quads is a line-based, plain text format, which represents one statement per line. Since Git is also treating lines as atoms on merging, it will automatically treat statements in N-Quads as atomic units. Further N-Quads in contrast to N-Triples supports the encoding of complete RDF datasets. N-Triples is a subset of N-Quads, by only using the default graph. Another candidate would be TriG (Turtle extended

[5]https://git-scm.com/
[6]https://github.com/about, 2016-05-04
[7]http://nvie.com/posts/a-successful-git-branching-model/
[8]https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow
[9]Concurrent Versions System, http://savannah.nongnu.org/projects/cvs
[10]https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/
[11]https://www.w3.org/TR/2014/REC-turtle-20140225/
[12]https://www.w3.org/TR/2015/NOTE-rdfa-primer-20150317/
[13]https://www.w3.org/TR/2014/REC-n-triples-20140225/
[14]https://www.w3.org/TR/2014/REC-trig-20140225/
[15]https://www.w3.org/TR/2014/REC-json-ld-20140116/
[16]https://www.w3.org/TR/2014/REC-n-quads-20140225/

by support for RDF datasets), in contrast to N-Quads one line doesn't necessarily represent one statement. Also due to the predicate and object list features (using ; or , as delimiter) as well as multi line literals, automatic line merges can destroy the syntax. Similar problems would occur with the other serialization formats listed above. To further ensure stability and comparability of the files we are using a canonicalized serialization.

Halilaj et al. [9] propose the usage of Turtle in Git repositories, to address the requirement to be editor agnostic. Since a transformation to any other serialization format is possible, e.g using rapper[17] or Jena RIOT[18], our approach doesn't put additional constraints on the usage of the serialization format in an editor application. Further as stated above, we find N-Quads to be of better fit for Git versioning than Turtle.

3.3 Blank Nodes in Versioning

Using RDF as an exchange format, still blank nodes are a problem we have to deal with. Blank nodes are identifiers with a local scope and so might be different for each participating platform. Implementing all atomic operations using atomic graphs as proposed in [3] and also discussed in [13] could be a solution, by building an identity for the blank nodes using their context. Still this would involve additional effort and wouldn't directly lead to a practical and working solution for a distributed collaboration system. Thus we have decided to follow the recommendation of RDF 1.1, which recommends replacing blank nodes with IRIs [6], and assume that each graph managed by our store is skolemized in advance.

4. METHODOLOGY

In this chapter we are describing our system and methodology. The system and methodology is structured in three steps: (1) the read and write interface using SPARQL 1.1 Select and Update, (2) the translation of the operations of the read/write interface to the respective versioning operations on the Git repository, and (3) the stage making use of the Git system to enable collaboration workflows. Each step is supported by our implementation as described later.

4.1 SPARQL 1.1 Read/Write Interface

As an interface accessible to other applications, we are using a SPARQL 1.1 endpoint. The endpoint supports SPARQL 1.1 Select and Update to provide a read/write interface on the RDF data. The Select Queries can be used to read data from the underlaying store and the Update Queries to add, change or delete data from the store. In the following example we want to show how an incoming Update Query will affect the file system. Therefor the Quit Store contains two named graphs `http://dbpedia.org/` serialized in a file `dbpedia.nq` (cf. listing 1) and `http://my.quit.graph/` with the corresponding file `default.nq` (cf listing 2).

```
<http://dbpedia.org/resource/Aachen> <http://www.w3.org↵
  /1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/↵
  ontology/City> <http://dbpedia.org/> .
```

Listing 1: An example graph with one statement from DBpedia (`dbpedia.nq`)

[17]http://librdf.org/raptor/rapper.html
[18]https://jena.apache.org/documentation/io/

```
<http://subject1> <http://predicate1> <http://object1> <http↵
  ://my.quit.graph/> .
```

Listing 2: A simple graph containing one statement (`default.nq`)

As mentioned above, the Quit Store accepts Select and Update Queries, whereby the Update operation may result in a new version of one or more named graphs depending on the patterns used. Listing 3 shows an Update Query executed on the Quit Store. The query moves all triples of the named graph `http://dbpedia.org/` and inserts these into the named graph `http://my.quit.graph/` and the differences between the resulting files can be seen in listings 4 and 5.

```
INSERT {
  GRAPH <http://my.quit.graph/> { ?s ?p ?o }
} WHERE {
  GRAPH <http://dbpedia.org/> { ?s ?p ?o }
};
DELETE {
  GRAPH <http://dbpedia.org/> { ?s ?p ?o }
} WHERE {
  GRAPH <http://dbpedia.org/> { ?s ?p ?o }
}
```

Listing 3: An example SPARQL 1.1 Update query

```
--- a/dbpedia.nq
+++ b/dbpedia.nq
@@ -1,4 +0,0 @@
-<http://dbpedia.org/resource/Aachen> <http://www.w3.org↵
  /1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/↵
  ontology/City> <http://dbpedia.org/> .
```

Listing 4: The differences of the file `dbpedia.nq` before and after the query execution

```
--- a/default.nq
+++ b/default.nq
@@ -1,3 +1,7 @@
+<http://dbpedia.org/resource/Aachen> <http://www.w3.org↵
  /1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/↵
  ontology/City> <http://my.quit.graph/> .
 <http://subject1> <http://predicate1> <http://object1> <http↵
   ://my.quit.graph/> .
```

Listing 5: The differences of the file `default.nq` before and after the query execution

4.2 Translate Read/Write to Git

The SPARQL 1.1 read/write interface allows users to query and edit RDF data contained in the quad store. Since we want to enable a collaboration system using Git, these pure read/write operations have to be transformed to operations on the Git repository. A read operation will not cause direct changes on the Git repository and thus the write operation is of most interest here. The major operations on a versioning system are *commit* to create a new version in the version log and *merge*, and *revert* as operations on the version log. This task is taken by a *Query-Analyzer* in combination with the *Quad-Store* interface.

The *commute* operation mentioned in [5] is not of a high importance for a distributed version control system like Git, since it can deal with multiple parallel branches. Thus it

doesn't need to merge branches by commuting them into a linear version history, as it is shown in [5]. Nevertheless the *commute* operation can still be executed in Git using the interactive *rebase* operation. In the following we are describing the individual operations on the versioning system and their execution.

Commit.

If the store receives an Update Query via its API, we execute this query and serialize and canonicalize the named graphs and write them to their corresponding files. We do not need to check whether files have changed since the previous commit, because we can use the `git add` command with the `--update` parameter, which will add all changed files under version control to the staging area. Note that due to the canonicalization two equal graphs will also result in the same serialization and if the staging area is empty, nothing has to be commited to the version log. If after adding the files the staging area isn't empty a `git commit` will succeed and a new commit is created in the Git version log. Figure 1 depicts an initial commit "A" without any predecessor resp. parent commit and a commit "B" referring to its parent "A".

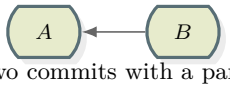

Figure 1: Two commits with a parent relation

Formally speaking a version of a graph produced with Quit Store is: Let G be a graph under version control in a Quit Store Q and F_G a serialized representation of G in a file. $B_{\{A\}}(\{F_G\})$[19] is a commit containing the file F and referring to its parent commit A (cf. fig. 1). G' will be the new version of G after any change operation regarding G was executed on Q, it will result in $F_{G'}$ with $F_G \neq F_{G'}$. $F_{G'}$ is the new version of the file added to the new commit $C_{\{B_{\{A\}}\}}(\{F_{G'}\})$. Since a commit is referring to its predecessor and not vice versa, nothing hinders us from creating a second commit $D_{\{B_{\{A\}}\}}(\{F_{G''}\})$, which is then a new *branch* or *fork*, which is diverged from B. Taking the commits A, $B_{\{A\}}$, $C_{\{B\}}$, and $D_{\{B\}}$ results in a directed rooted in-tree, as depicted in fig. 2.

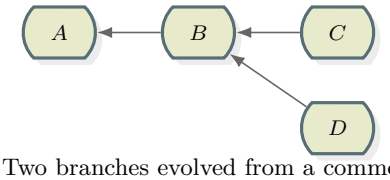

Figure 2: Two branches evolved from a common commit

Based on the comparison of commits $C(\{F_G\}), D(\{F_{G'}\})$ and their respective contents $(F_G, F_{G'})$ a patch can be generated by calculating the difference $\Delta(F_G, F_{G'})$ between the contents. A patch is a pattern that contains lines that have to be deleted from a file and lines that have to be inserted into a file, in order to transform the version F_G of the file into the version $F_{G'}$. Applying this method on canonicalized N-Quads files, will give us a patch containing one triple per line, which has to be inserted resp. deleted from the graph, thus $\Delta(F_G, F_{G'}) \Leftrightarrow \Delta(G, G')$. In our case a patch is every change of a version controlled file in the used Git repository.

[19] In the further writing, the indices and arguments of commits are sometimes omitted for better readability, while clarity should still by maintained by using distinct letters

Since we are receiving the change operations from the top layer SPARQL endpoint, note that there is an important difference between a received Update Query with `INSERT DATA` or `DELETE DATA` operation or even `INSERT … WHERE` resp. `DELETE … WHERE` and the resulting patch after applying the changes. There might be a statement a user wants to be deleted that doesn't exist or a user wants to add a statement that is already contained in the store. In turn in general a valid patch is a patch that can be expressed as a valid SPARQL 1.1 Update Query using `INSERT DATA` and `DELETE DATA` operations, where the *quad data* of the `DELETE DATA` contains all removed statements and the *quad data* of the `INSERT DATA` all added statements from the patch.

Merge Different Branches.

If the tree of commits is diverged, as shown in the example of fig. 2 we now want to merge the branches again. This allows us to get a version of the graph, containing changes made in the different branches. This is done by creating a commit $E_{\{C_{\{B\}}, D_{\{B\}}\}}(\{F_{G'''}\})$, which has two predecessor commits it is referring to. Taking the commits A, $B_{\{A\}}$, $C_{\{B\}}$, $D_{\{B\}}$, and $E_{\{C,D\}}$, we get an acyclic directed graph, as it is depicted in fig. 3.

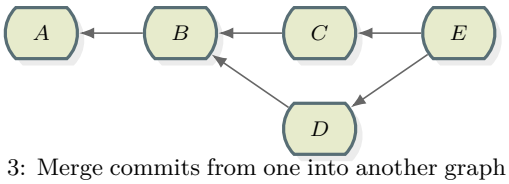

Figure 3: Merge commits from one into another graph using `git merge`

Since we are not only interested in the branching and merging model of the commits, we want to know, what a merge means for the graph G''', resulting from merging G' and G''. Note, that merging in this context is not to be understood as in the RDF 1.1 Semantics Recommendation [10] as the union of two graphs. Git is using a three-way-merge, taking into account the versions of the files in the two commits to be merged $F_{G'}$, $F_{G''}$ and the most recent common ancestor F_G[20]. For each line Git decides on a merge commit whether it will be included into the result according to the decision matrix given in table 1. Since we are using N-Quads files, the decisions will be the same for triples contained in the graphs G', G'', and G. A merge conflict in Git occurs, when two close by lines where added or removed in different files and thus Git can't decide, whether they are resulting from the same original line or if they are contradicting. For our RDF data model these merge commits can be easily resolved, by deciding for each line resp. triple based on table 1 and sorting the resulting lines alphabetically to maintain the canonicalization. Semantic contradictions within the resulting RDF graph can then be dealt with using additional tools, which can work on valid RDF files from this point on.

Revert a Commit.

Reverting the commit $B_{\{A\}}(\{F_G\})$ and directly applying it to B is done by creating a commit $B^{-1}_{\{B\}}(\{F_{\tilde{G}^0}\})$ (where $A(\{F_{G^0}\})$). Where one gets $F_{\tilde{G}^0}$ by calculating the patch between $\Delta(F_{G^0}, F_G)$, swapping the set of added and deleted

[20] How does Git merge work: https://www.quora.com/How-does-Git-merge-work, 2016-05-10

A $F_{G'}$	B $F_{G''}$	base F_G	result $F_{G'''}$	
O	O	O	O	Non existing lines
X	X	X	X	Lines existent in all files will also be in the result
X	O	O	X	A line added to $F_{G'}$ is also added to the result
O	X	O	X	A line added to $F_{G''}$ is also added to the result
O	X	X	O	A line removed from $F_{G'}$ is also not added to the result
X	O	X	O	A line removed from $F_{G''}$ is also not added to the result
X	X	O	X	A line added to both branches is also added to the result
O	O	X	O	A line removed from both branches is also not added to the result

Table 1: Decision table for the different situations on a three-way-merge (X = line exists, O = line doesn't exist)

lines, and applying the patch to F_G, which results in $F_{\tilde{G}^0}$. After this operation $F_{\tilde{G}^0} = F_{G^0}$ and thus $\tilde{G}^0 = G^0$.

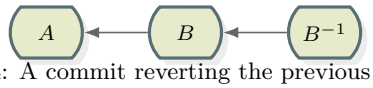

Figure 4: A commit reverting the previous commit

Using Git, reverting a commit is possible via the command `git revert <commitid>`. Figure 4 shows a versioning log containing three commits, where the latest commit reverts its parent and thus the state of the working directory is equal to the one produced by the first commit. While it is obvious, how to revert the previous commit it may be a problem, if other commits exist between the commit to be reverted and the current top of the versioning log (*HEAD*). In this case again a three-way-merge is applied (cf. table 1), where the merge *base* is the commit to be reverted, branch A is the parent commit of the commit which is to be reverted, and branch B the current *HEAD*. Merge conflicts can be resolved in the same way, as for merge commits.

4.3 Enable Collaboration Workflows With Git

So far all operations where executed on a local versioning graph, which can diverge and be merged, but still no collaboration with remote participants is possible. To allow collaboration on the World Wide Web, the versioning graph can be published (*push*) to a remote repository, from where other collaborators can copy (*clone*) the complete graph. If a collaborator already has cloned a previous version of the versioning graph, she can update here local graph by executing a *pull*. Further it is possible for collaborators to create their own branches, which they don't frequently merge with the branches of other collaborators, or which is merged in a later stage, e.g. for an independent development of an RDF dataset; this is called *fork*.

In the domain of software development different workflows have evolved in using Git to improve the quality of collaboratively developed software [17, 14] e.g. *Gitflow*[21] and the *Forking Workflow*[22]. Halilaj et al. [9] have proposed a branching model for RDF vocabulary development, which is built on top of the *Gitflow* model.

[21]http://nvie.com/posts/a-successful-git-branching-model/
[22]https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow

5. IMPLEMENTATION

The prototypical implementation of the Quit Store[23] is developed with Python[24] with the RDFlib[25] to deal with RDF and combined with Flask API[26] to provide a *SPARQL 1.1 Interface* via HTTP (cf. fig. 5). The underlaying storage of the RDF dataset is implemented by an in memory *Quad-Store* and local files which are kept in sync with the corresponding named graphs in the store (cf. *File References* in fig. 5). Every file contains alphabetically sorted N-Quads.

Incoming queries are analyzed by the *Query-Analyzer*, which distinguishes between SPARQL Update and Select Queries. The store also contains a separate graph for configuration settings, which is shown in listing 6. The configuration graph is not part of the store which is accessible via the SPARQL endpoint.

```
@base <http://quit.aksw.org/> .
@prefix conf: <http://my.quit.conf/> .

conf:dbpedia a <Graph> ;
  <graphUri> <http://dbpedia.org/> ;
  <isVersioned> 1 ;
  <hasQuadFile> "dbpedia.nq" .

conf:graph1 a <Graph> ;
  <graphUri> <http://my.quit.graph/> ;
  <isVersioned> 1 ;
  <hasQuadFile> "graph.nq" .
```

Listing 6: An example configuration for a Quit Store

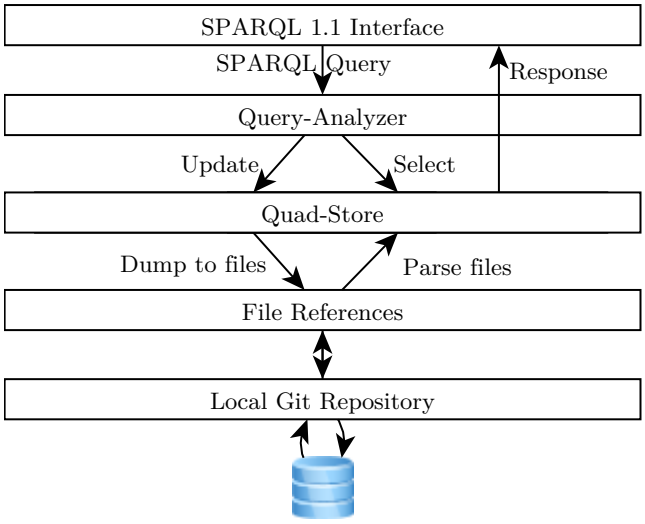

Figure 5: The components of the Quit Store

To create a new version of a graph on the local file system we have to update the content of the local files. Therefor the `FileReference` class provides methods to get and set the file content, as well as canonicalizing each file using `cant.py` from the Semantic Web Application Platform - SWAP[27].

[23]https://github.com/AKSW/QuitStore
[24]https://www.python.org/
[25]https://rdflib.readthedocs.io/en/stable/
[26]http://flask.pocoo.org/
[27]https://www.w3.org/2000/10/swap/

After an execution of a Select Query the files and the store remain in the same state. If we execute an Update Query, all named graphs are serialized to the corresponding files and new versions are staged and commited by the methods of the `GitRepo` class, which manages the local Git repository.

The default Git operations *pull*, *checkout* and *push* are also available via the Quit Store HTTP interface. They allow the user to navigate in the version history and synchronize with remote repositories.

Write Through Strategies.

Before an operation is completed and the user can get a success message, the store has to make sure to persist the effective changes. In order to allow a flexible tradeoff we provide configurable multistage write through strategies and levels of failures:

Query-Receive Fails if the store is not available (SPARQL interface)

Query-Execution Fails on syntax errors, or if a graph is not available (Query-Analyzer, Quad-Store)

Push (`git push`) Fails on remote conflicts, or if the repository server is not available

Depending on the desired use case there might be different requirements on availability resp. request time outs and consistency. Based on these requirements the necessary write through strategy can be selected.

6. RELATED WORK

As related work we consider both: approaches for versioning of RDF data, and means for collaborating on RDF data by synchronizing or exchanging dataset differences. Versioning of RDF data is a long standing topic. Berners-Lee and Connolly [13] introduce an ontology that describes patches in "a way to uniquely identify what is changing" and "to distinguish between the pieces added and those subtracted". This can be used for subsequently expressing the evolution of an RDF dataset, e. g. using the *Quit Diff* [2] tool, which allows to generate patches by comparing graph serializations or commits in a Git repository and expressing the changes using changeset vocabularies.

Beside syntactical diffs there are important approaches of ontology evolution and versioning using description logic provided with OWL[19]. In [20] the authors describe two tools they developed to target this problem. The first named *owl2diff*[28] detects changes between different versions of OWL ontologies and the second *owl2merge* helps "to resolve conflicts and perform a three-way merge". We tested the `owl2diff` tool with the example described in section 4.1 by transforming the resulting files to Turtle. The result is shown in listings 7 and 8.

```
* OntologyFormat("KRSS2 Syntax")
- Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
- Prefix(rdf:=<http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
- Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
- Prefix(:=<http://www.semanticweb.org/owl/owlapi/turtle#>)
- Prefix(xml:=<http://www.w3.org/XML/1998/namespace>)
- Prefix(rdfs:=<http://www.w3.org/2000/01/rdf-schema#>)
```

[28]https://github.com/utapyngo/owl2vcs

```
- ClassAssertion(<http://dbpedia.org/ontology/City> <http://↵
  dbpedia.org/resource/Aachen>)
```

Listing 7: The differences detected by `owl2diff` of the file `dbpedia.ttl` and …

```
+ ClassAssertion(<http://dbpedia.org/ontology/City> <http://↵
  dbpedia.org/resource/Aachen>)
```

Listing 8: … of the file `default.ttl`

Other systems, such as OWLDiff [12] and Ecco [8] also present tools for comparing two versions of an OWL ontology. In contrast to our approach these systems are assuming an expressive OWL ontology and are not focusing on working with general RDF datasets.

Auer and Herre [3] suggest a versioning and evolution framework for RDF knowledge bases. This work provides a practical approach for dealing with blank nodes in change sets, and introduce a hierarchical system for structuring a set of changes, as well as a system to formalize the evolution patterns leading to the changes of a knowledge base. Cassidy and Ballantine [5] also discuss a version control system for RDF graphs. Their approach is based on a system called *Darcs* and covers the versioning operations *commute*, *revert* and *merge*. Unfortunately these approaches are not usable for distributed collaboration since they don't include a branching and merging model, which can easily deal with different co-existing versions, as it is possible with Git.

R&Wbase [16] is a tool which also has an understanding of coexisting branches within a versioning graph, which is very close to the concept of Git. In contrast to Git, R&Wbase always stores the differences between versions, rather then always the complete file at a certain version as it is done for Git. The individual change sets of the versions are stored in named graphs, this makes it impossible to use the system to manage RDF datasets with multiples named graphs.

The Git4Voc, as proposed by Halilaj et al. [9] is a methodology and collection of best practices for collaboratively creating RDF vocabularies using Git repositories. To support vocabulary authors in the process of creating rdf/owl vocabularies Git4Voc as implemented in VoCol as pre- and post-commit hooks, which call various tools for automatic checking the vocabulary specification. Even though Git4Voc is focusing on vocabularies and only adds a collection of pre- and post-commit hooks, it has formulated very important requirements for collaboration on RDF data. We have partially incorporated these requirements in section 2.

7. CONCLUSION & FUTURE WORK

In this paper we have presented a methodology together with the Quit Store tool for providing a SPARQL 1.1 read-/write interface to query and change an RDF dataset in a quad store. The contents of the store are tracked for version control in a local filesystem in parallel. These files are then managed using the Git distributed version control system to provide the versioning operations *commit*, *merge* and *revert*. By using the distributed features of Git, it is on the one hand possible to keep track and to integrate data provided by others and on the other hand possible to provide a highly dynamic collaboration platform. This brings us further on answering the question, how the collaborative curation of

distributed Linked Data Knowledge Bases can be synchronized. In future work, we have to evaluate this methodology and system regarding its correctness and usability with interested collaborating communities, as pointed out in section 1, and its performance and scalability regarding the size of datasets, the amount of change operations and the number of collaborating parties.

In the future the usage in enterprise scenarios as described in [7] is possible. We are also planning to lift the Structured Feedback protocol [1] to a next level by directly recording the user feedback as commits in a Quit Store, which can enable mighty co-evolution strategies. As there is a big ecosystem of methodologies and tools around Git for supporting the software development process, the Quit Store can help to create such an ecosystem for RDF dataset creation as well.

8. ACKNOWLEDGEMENTS

This work was partly supported by the following grants from the German Federal Ministry of Education and Research (BMBF) for the LEDS Project under grant agreement No 03WKCG11C and the European Union's Horizon 2020 research and innovation programme for the SlideWiki Project under grant agreement No 688095.

9. REFERENCES

[1] N. Arndt, K. Junghanns, R. Meissner, P. Frischmuth, N. Radtke, M. Frommhold, and M. Martin. Structured feedback: A distributed protocol for feedback and patches on the web of data. In *Proceedings of the Workshop on Linked Data on the Web co-located with the 25th International World Wide Web Conference (WWW 2016)*, volume 1593 of *CEUR Workshop Proceedings*, Montréal, Canada, Apr. 2016.

[2] N. Arndt and N. Radtke. Quit diff: Calculating the delta between rdf datasets under version control. In *12th International Conference on Semantic Systems Proceedings*, SEMANTiCS '16, Leipzig, Germany, Sept. 2016.

[3] S. Auer and H. Herre. A versioning and evolution framework for RDF knowledge bases. In *Proceedings of at Sixth International Andrei Ershov Memorial Conference - Perspectives of System Informatics (PSI'06), 27-30 June, Novosibirsk, Akademgorodok, Russia*, volume 4378, June 2006.

[4] G. Carothers. Rdf 1.1 n-quads: A line-based syntax for rdf datasets. https://www.w3.org/TR/2014/REC-n-quads-20140225/, Feb. 2014.

[5] S. Cassidy and J. Ballantine. Version control for RDF triple stores. In J. Filipe, B. Shishkov, and M. Helfert, editors, *ICSOFT 2007, Proceedings of the Second International Conference on Software and Data Technologies*, pages 5–12, Barcelona, Spain, 2007. INSTICC Press.

[6] R. Cyganiak, D. Wood, and M. Lanthaler. Rdf 1.1 concepts and abstract syntax. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/, Feb. 2014.

[7] M. Frommhold, N. Arndt, S. Tramp, and N. Petersen. Publish and Subscribe for RDF in Enterprise Value Networks. In *Proceedings of the Workshop on Linked Data on the Web co-located with the 25th International World Wide Web Conference (WWW 2016)*, 2016.

[8] R. S. Gonçalves, B. Parsia, and U. Sattler. Ecco: A hybrid diff tool for owl 2 ontologies. In P. Klinov and M. Horridge, editors, *OWLED*, volume 849 of *CEUR Workshop Proceedings*, 2012.

[9] L. Halilaj, I. Grangel-González, G. Coskun, and S. Auer. Git4voc: Git-based versioning for collaborative vocabulary development. In *10th International Conference on Semantic Computing*, Laguna Hills, California, Feb. 2016.

[10] P. J. Hayes and P. F. Patel-Schneider. Rdf 1.1 semantics. https://www.w3.org/TR/2014/REC-rdf11-mt-20140225/, Feb. 2014.

[11] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, and R. Cornelissen. Databugger: A test-driven framework for debugging the web of data. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion*, pages 115–118, Republic and Canton of Geneva, Switzerland, 2014.

[12] P. Kremen, M. Smid, and Z. Kouba. Owldiff: A practical tool for comparison and merge of owl ontologies. In F. Morvan, A. M. Tjoa, and R. Wagner, editors, *DEXA Workshops*, pages 229–233. IEEE Computer Society, 2011.

[13] T. B. Lee and D. Connolly. Delta: an ontology for the distribution of differences between rdf graphs. Technical report, W3C, 2001.

[14] S. Phillips, J. Sillito, and R. Walker. Branching and merging: an investigation into current version control practices. In *In International workshop on Cooperative and human aspects of software engineering, CHASE '11, ACM*, pages 9–15, 2011.

[15] T. Riechert and F. Beretta. 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] M. V. Sande, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens, and R. V. de Walle. R&wbase: git for triples. In C. Bizer, T. Heath, T. Berners-Lee, M. Hausenblas, and S. Auer, editors, *LDOW*, volume 996 of *CEUR Workshop Proceedings*, 2013.

[17] E. Shihab, C. Bird, and T. Zimmermann. The effect of branching strategies on software quality. *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 0:301–310, 2012.

[18] The W3C SPARQL Working Group. Sparql 1.1 overview. https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/, Mar. 2013.

[19] W3C OWL Working Group. OWL 2 Web Ontology Language document overview. https://www.w3.org/TR/2012/REC-owl2-overview-20121211/, Dec. 2012.

[20] I. Zaikin and A. Tuzovsky. Owl2vcs: Tools for distributed ontology development. In *Proceedings of the 10th International Workshop on OWL: Experiences and Directions (OWLED 2013) co-located with 10th Extended Semantic Web Conference (ESWC 2013)*, volume 1080 of *CEUR Workshop Proceedings*, Montpellier, France, May 2013.