Towards Versioning of Arbitrary RDF Data

Marvin Frommhold
Agile Knowledge Engineering and Semantic Web
Institute of Computer Science
University of Leipzig, Germany
frommhold@informatik.uni-leipzig.de

Rubén Navarro Piris
eccenca GmbH
Hainstr. 8
04109 Leipzig, Germany
ruben.navarro.piris@eccenca.com

Natanael Arndt
Agile Knowledge Engineering and Semantic Web
Institute of Computer Science
University of Leipzig, Germany
arndt@informatik.uni-leipzig.de

Sebastian Tramp
eccenca GmbH
Hainstr. 8
04109 Leipzig, Germany
sebastian.tramp@eccenca.com

Niklas Petersen
Enterprise Information Systems
Institute for Applied Computer Science
University of Bonn, Germany
petersen@cs.uni-bonn.de

Michael Martin
Agile Knowledge Engineering and Semantic Web
Institute of Computer Science
University of Leipzig, Germany
martin@informatik.uni-leipzig.de

ABSTRACT

Coherent and consistent tracking of provenance data and in particular update history information is a crucial building block for any serious information system architecture. Version Control Systems can be a part of such an architecture enabling users to query and manipulate versioning information as well as content revisions. In this paper, we introduce an RDF versioning approach as a foundation for a full featured RDF Version Control System. We argue that such a system needs support for all concepts of the RDF specification including support for RDF datasets and blank nodes. Furthermore, we placed special emphasis on the protection against unperceived history manipulation by hashing the resulting patches. In addition to the conceptual analysis and an RDF vocabulary for representing versioning information, we present a mature implementation which captures versioning information for changes to arbitrary RDF datasets.

Keywords

RDF, Versioning, Blank Node, RDF Quad, Hashing

1. INTRODUCTION

In recent years, the need for shared semantics and integration of heterogeneous data across enterprises has dramatically increased. The Semantic Web is on its way to provide tools and technologies [18] to express a shared meaning of data that is crucial for distribution and integration of data. In the LUCID research project[1], we focus on semantic technologies which allow for distribution of RDF data in decentralized value chain networks in order to make the flow of information more efficient, more effective and more robust. In addition to that, within the LEDS research project[2], we concentrate on the co-evolution of datasets outside and inside of an enterprise to allow for better integration of public and private data. Both usage scenarios strongly depend on a system which keeps track of changes of RDF datasets and thereby enables the collaborative development of RDF data in distributed environments.

An area with similar requirements is software development where versioning mechanisms have been successfully applied [1]. Unfortunately, there is a lack of robust and highly efficient versioning systems for the Resource Description Framework (RDF) [9], hindering a wide adoption in enterprises.

In this paper, we present a Version Control System (VCS) for arbitrary RDF data to overcome this gap. It creates a patch for each SPARQL Update query [10] containing the added (addition set) and deleted (deletion set) triples and their corresponding RDF graphs, as well as provenance information such as the author of the operation and a change reason.

In particular, we present the following main contributions of our proposed VCS:

- Change detection across multiple RDF graphs enabling support for datasets consisting of any number of graphs.
- Full support for versioning of blank nodes allowing the deployment of the system without the need for preprocessing the target datasets.
- A vocabulary to describe changes to an RDF dataset allowing for distribution of patches.
- Protection against unperceived manipulation of patches.

SEMANTiCS 2016, September 12 - 15, 2016, Leipzig, Germany
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4752-5/16/09...$15.00
DOI: http://dx.doi.org/10.1145/2993318.2993327

[1]http://www.lucid-project.org/
[2]http://leds-projekt.de

This forms the foundation for a versioning of arbitrary RDF data enabling efficient data exchange in distributed networks.

The paper is structured as follows: In section 2 we give an overview of the requirements for a versioning system for arbitrary RDF data. A comparison of our approach to related work is discussed in section 3. Subsequently, we present the concepts of our approach in section 4, followed by an overview of the implementation in section 5. An evaluation of the system is presented in section 6 and we conclude with an outlook on future work in section 7.

2. REQUIREMENTS & PRELIMINARIES

First we formulate some requirements and preliminaries for a VCS that is suitable to track changes made to any possible RDF dataset.

Based on the features of traditional file-based VCS it has to support basic versioning operations such as the reversal of a change, the rollback to a specific version and the merge of diverged versions, which is likely to happen in a distributed environment. Provenance plays an important role in enterprise applications, therefore the reason and author of a change have to be tracked as well.

The applicability for versioning of arbitrary RDF data yields to additional requirements. Obviously, to be widely adopted in enterprises, such a system must be deployable without the need to make changes to the versioned dataset. In addition to support for RDF datasets consisting of a collection of RDF graphs, another core feature of the RDF standard, the concept of blank nodes, must be supported. Blank nodes are problematic in terms of matching and addressability, especially when working with OWL ontologies.

To illustrate our discussion, we use the RDF dataset shown in Listing 1 as a running example for the rest of the paper. The dataset consists of one named graph containing two OWL classes with restrictions typically involving blank nodes. All namespaces used in the examples are the ones returned by looking them up on prefix.cc[3].

```
ex:GraphA {
  ex:ClassA a owl:Class ;
    rdfs:subClassOf [
      owl:onProperty ex:someProperty ;
      owl:cardinality "1"
    ] .
  ex:ClassB a owl:Class ;
    rdfs:subClassOf [
      owl:onProperty ex:someProperty ;
      owl:cardinality "1" ;
      owl:someValuesFrom ex:ClassC
    ] .
}
```

Listing 1: RDF dataset in TriG [4] syntax used as a running example.

In the following we elaborate the requirements of the LUCID and LEDS research projects.

[3]`http://prefix.cc/PREFIX.file.txt`, replace `PREFIX` with the prefix to look up

2.1 Invertible Patches

Cassidy and Ballantine [7] propose a semi-formal model for RDF patches based on the *theory of patches* of Darcs[4]. In their work, a version is a sequence of patches and a patch consists of two sub-graphs, one to be added and the other to be deleted. To manipulate these sequences, they describe three basic operations:

- *Commute*: This operation swaps the order of two consecutive patches. The commute fails if a conflict arises [7, Section 3.1].
- *Revert*: This operation reverts the most recent patch from the current sequence. With the help of the commute operation it is possible to revert a patch that is not at the head of a sequence.
- *Merge*: This operation combines two patches that apply to the same sequence. A merge is possible if the commute operation between the two patches succeeds.

According to [7] a patch must be invertible in order to allow these operations. They define the inverse of a patch as the simplest patch to achieve this. By their definition, the application of a patch followed by its inverse restores the state before the patch. In the context of RDF, the inversion of a patch can be achieved by swapping the add and delete sub-graphs, hence removing all added statements and putting back all deleted statements by an update request.

2.2 Change Detection

As a patch must be invertible, this yields to a particular requirement for the tracking of changes. Let us consider a SPARQL Update `INSERT DATA` query which adds the triple `ex:ClassA a owl:Class` to the graph of our running example in Listing 1. According to the RDF specification, when adding a triple already contained in a graph, nothing should happen. That means the state of the graph and therefore the dataset will not change. If a VCS only tracks the executed SPARQL Update queries (query log), the resulting patches would be not invertible. When reverting the patch based on that query, a naive approach would be to execute a SPARQL Update `DELETE DATA` query which removes that triple from the graph leading to a state not equal to the state before the patch. As noted before, reverting a patch must result in the same state as before the patch was applied. Taking this into account, a VCS must be able to detect and track actual changes to the dataset in order to create invertible patches.

2.3 Blank Nodes

As stated in an empirical survey of Mallea et al. [17], over 50% of published datasets contain blank nodes. This emphasizes the need for blank node support in a VCS.

The standard semantics interprets blank nodes as existential variables, denoting the existence of some unnamed resource. This leads to difficulties when it comes to versioning of datasets containing blank nodes. To illustrate the issue, let us consider the SPARQL Update query in Listing 2. The query adds the triple `[] owl:someValuesFrom ex:ClassC` to the restriction of `ex:ClassA` making it equivalent to the restriction of `ex:ClassB`. To revert this change, the triple has to be deleted again. As SPARQL does not allow blank

[4]http://darcs.net/Theory

nodes in a DELETE DATA query[5], the only way to remove the triple is using a DELETE query in conjunction with a WHERE clause addressing the blank node. Therefore a VCS must additionally track the context of any blank nodes contained in a triple to be able to address them in subsequent operations or other RDF repositories containing a fork of the versioned dataset.

```
INSERT {
  ?restriction owl:someValuesFrom ex:ClassC .
}
USING ex:GraphA
WHERE {
  ex:ClassA rdfs:subClassOf ?restriction .
  ?restriction owl:onProperty ex:someProperty .
}
```

Listing 2: SPARQL Update query adding a triple containing a blank node to the graph of the example dataset in Listing 1.

2.4 RDF Quads

Let us consider a SPARQL Update query that adds a triple to a named graph G[6] of an RDF dataset consisting of multiple graphs. If the patch tracks only the triple without the graph reference, it becomes not invertible, as the graph to delete the triple from can not be determined. That means, to support all possible RDF datasets, a VCS needs to track the corresponding RDF graph of a triple, known as quad [8].

2.5 Version Integrity

Provenance and trust play an important role when working with a VCS. This includes mainly ensuring the integrity of a patch, protection against manipulation of patches as well as versions (sequences of patches). When distributing a patch, it should be possible to verify the integrity of this patch. Optionally, a digital signature of the patch can be provided lowering the barrier of trust [6]. That means, the system must provide an algorithm to produce a unique hash for RDF data in the presence of quads and blank nodes. A patch should include the hash of its predecessors preventing unperceived history manipulation.

2.6 Change Representation

To be able to store the changes, a VCS requires a suitable representation format for its patches. Such a representation must reference the changes to an RDF dataset as well as additional meta-information such as the author, change reason and time stamp to provide provenance. As a VCS is typically used in distributed environments, it is crucial to allow for sharing of patches. It must be possible to apply a patch to an RDF repository, whether it is the same or another repository containing a fork of the same dataset.

[5]http://www.w3.org/TR/2013/REC-sparql11-query-20130321/#sparqlGrammar
[6]assuming G is not the default graph (http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/#dfn-default-graph)

3. RELATED WORK

Various different approaches for versioning of RDF have been published. Since special requirements are placed on data management in enterprise environments, they typically use databases to store their data. For this reason, we now only consider previous work introducing version control for RDF repositories (triple or quad stores) and therefore supporting query languages like SPARQL to query and update the data. Table 1 provides a feature comparison between the related work and our work.

In 2002, as part of the On-To-Knowledge project, Kiryakov and Ognyanov [14] present an ontology middleware tracking changes in an RDF repository. They are the first to introduce a triple-based versioning system. Saving the meta-information in RDF allows them to query, manage and edit this information using the same technologies as for the data itself. To represent the meta-information they also provide a vocabulary for tracking, versioning, security and user-defined information. The RDF 1.0 specification [15] became a recommandation in 2004, specifying any expression in RDF as a collection of triples. This fact would explain why Kiryakov and Ognyanov have not considered quads at all (they define a repository as a set of triples). As blank nodes are not mentioned we assume they are not supported, too.

Five years later, in 2007, Auer and Herre [2] describe a system to support the evolution of RDF ontologies. They are the first to consider blank nodes which typically occur in ontologies and define an *atomic graph* as the smallest unit of change that cannot be divided without duplicating blank nodes. A patch consists of a set of these atomic graphs added to or deleted from an RDF graph. However, their main focus is to aggregate these patches into higher level aggregations and classification into ontology evolution patterns. The derived ontology evolution patterns together with data migration algorithms are used to allow automatic data migration in distributed environments. This aggregation makes it difficult, for example, to revert a single atomic change at a later time. The term of an RDF dataset was officially introduced by the RDF 1.1 specification [9] in the year 2011. This can be an indicator why Auer and Herre are focusing on versioning of a single RDF graph, too.

Also published in 2007, Cassidy and Ballantine [7] propose a solution based on a semi-formal model named the *theory of patches* of Darcs, where a version is a sequence of patches. They define that a patch consists of two subgraphs, one containing the effectively added and the other one the effectively deleted triples. While they only use this basic form of a patch, they note possible meta-information can be attached to a patch if necessary. Their main contribution is the transformation of the basic versioning operations of Darcs, a traditional file-based VCS, to RDF. They are the first to define requirements a patch must fulfill to support these operations. Unfortunately, they only consider versioning for triples and do not support blank nodes. However, they note that their approach is entirely compatible with the formulation of [2] which means the described RDF versioning operations are able to support blank nodes.

About six years later, Vander Sande et al. [21] present *R&Wbase*, storing triples as consecutive deltas to lower the storage footprint when versioning RDF data. A delta has meta-information associated, including a hash value, enabling provenance. Vander Sande et al. were the first to introduce hashing for patches, but unfortunately they have not described the hashing algorithm. Because their interpretation layer hides the graphs and exposes a triple-based repository, it is not possible to use this approach to version an RDF dataset. Their approach provides naive blank node support: either delete all triples matching a triple pattern, or delete a specific triple if the internal identifier of the repository is used to reference contained blank nodes. This fact of using the internal identifiers of the repository makes it impossible to apply a patch to another repository.

In 2014, Graube et al. [12] promote *R43ples* acting as a proxy in front of any SPARQL service. Their work is the first to provide an approach allowing for versioning of RDF datasets, including a vocabulary to represent patches referencing the changed named graphs. While blank node support is not implemented, they propose Skolemization before executing a SPARQL query. Skolemizing the queries means that the versioned dataset must not contain blank nodes prior to deployment or the dataset has to be skolemized, too.

Finally, we can summarize that all of our requirements were discussed before, including descriptions of possible solutions, except for hashing of RDF. A comprehensive approach, however, was not presented before.

4. CONCEPTS

While none of the related work (section 3) meets our requirements, the proposed versioning system by Cassidy and Ballantine [7], based on a semi-formal model, appears to be the most suitable as groundwork for our approach. In the following we present solutions for the open requirements, namely blank nodes, hashing and quads.

4.1 Addressing Blank Nodes

As the approach of Cassidy and Ballentine does not support blank nodes, it suffers from the same problem regarding the addressability of blank nodes outside their original graph described in subsection 2.3. One may argue skolemizing the versioned dataset will solve the problem, but this is against the requirement of not changing the dataset (section 2). Skolemization of blank nodes changes the dataset and cannot be undone under certain circumstances [13]. This circumstance prevents the possibility to switch the VCS or even uninstall it completely.

To solve this, we use the idea of Tummarello et al. [20] introducing the formal definition of a *Minimum Self-Contained Graph* (MSG). The key concept of a MSG is to consider triples containing blank nodes always together with their surrounding triples when storing or transferring them. According to Tummarello et al. the MSG of a triple `s`, written as `MSG(s)`, is the set of triples containing `s` and recursively, for all blank nodes involved so far, the MSG of the triples containing these blank nodes. In particular each triple belongs to one and only one MSG, thus it makes no difference which triple is chosen as a starting point to build an MSG. The MSG of a triple with no blank nodes is therefore the triple itself. That means, applied to the creation of patches, whenever a triple containing blank nodes is deleted or added, the MSG of that triple will be referenced in the patch instead. For example, a patch as shown in Listing 3 based on the SPARQL Update query in Listing 2 would reference two MSGs: the old MSG that gets deleted containing the blank node of the triple `[] owl:someValuesFrom ex:ClassC` and the new MSG that gets added extended by this triple.

Because of the semantics of blank nodes it happens that an RDF graph may contain redundant information. Such non-lean graphs contain indistinguishable MSGs [19]. While the addition of indistinguishable MSGs is not a problem, the VCS must be aware of this when deleting such MSGs, for example during a revert of a patch. Our VCS ensures that only one of the indistinguishable MSGs is removed and no sub-graph of another MSG is removed by mistake due to isomorphism.

4.2 Hashing RDF

As the work of Carroll [6] shows, it is possible to calculate the hash for a large class of RDF graphs in $O(n\ log\ n)$ even in the presence of blank nodes. The key of his approach is the modification of the representation of an RDF graph without a change in its meaning before hashing. Although being rare [17], there might be situations where hashing results in a non-deterministic way, especially if many blank nodes with no property or label attached are involved. But as there is a finite number of possible serializations, it is possible to compute all hashes and treat them as equivalent.

4.3 Versioning Vocabulary

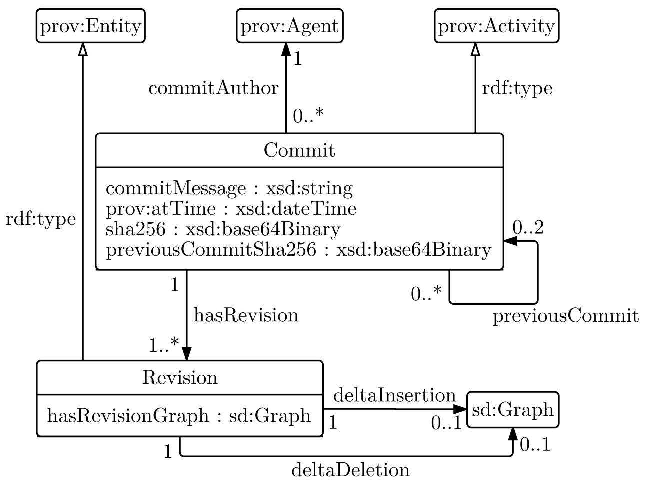

Figure 1: The revision vocabulary.

We have developed a revision vocabulary[7], shown in Figure 1, which is based on the structure of the Delta ontology [3] and reuses concepts of the PROV-O ontology [16]. By reusing concepts of PROV-O, a patch fulfills the notion of provenance of [11]. A patch includes a reference to the author, change reason, time stamp and changes of a transaction triggered by a SPARQL Update query. To meet the requirement of tracking changes of RDF quads (see subsection 2.4), we store the additions and deletions for all affected named graphs of a SPARQL Update query in the same single patch.

Another advantage of our vocabulary is the support for two different serialization formats. A triple based format

[7]https://vocab.eccenca.com/revision/

Work	Date	Invertible Patches	Change Detection	Blank Nodes	Quads	Hashing	Change Representation
[14]	2002	yes	yes	n/a	no	no	yes
[2]	2007	n/a	yes	yes	no	no	yes
[7]	2007	yes	yes	no	no	no	no
[21]	2013	n/a	n/a	partially	no	yes	no
[12]	2014	yes	yes	no	yes	no	yes
Our Work	2016	yes	yes	yes	yes	yes	yes

Table 1: Feature comparison of versioning approaches.

using reification[8] for situations where the data and versioning information should remain in the same repository without polluting it with too many graphs. Additionally, a quad based format is available which provides a lower memory footprint by using named graphs to store the additions and deletions. This format is preferred if the versioning information can be stored in a separate repository. Listing 3 illustrates how a patch looks like using the quad based format.

To increase trust, parties can optionally certify a patch by signing its hash value and attach the digital signature to the patch, for example using the Cert Ontology[9].

5. IMPLEMENTATION

We implemented our proposed concept as part of the LUCID endpoint which acts as a proxy in front of one or more RDF repositories. The current implementation works in conjunction with an OpenLink Virtuoso Universal Server[10]. To review the history of a versioned dataset, the LUCID endpoint provides an interface to retrieve patches by various parameters, for example, patches containing changes for a specific resource.

5.1 Architectural Overview

Figure 2 displays the structure and process flow of our VCS. During the first startup, the LUCID endpoint initializes the Virtuoso instance by adding triggers, procedures and a custom table (diff table) for change detection (1). Since changes are detected by the Virtuoso server, users are able to send SPARQL Update queries either through the LUCID endpoint interface (2a) or directly to the Virtuoso instance (2b). Each addition or deletion to the Virtuoso instance is saved to the diff table referencing the author, change message and transaction that triggered the change (3). Within the same transaction, the MSG calculation is performed for all blank nodes in the diff table. The triples of these MSGs are also added to the diff table (4). The LUCID endpoint then reads the triples and meta-information per transaction and creates the patch (5) followed by the hash calculation (6). Finally, the patch is stored depending on the configuration either in the same Virtuoso instance (7a) or in another repository (7b).

Additionally, we want to note that we implemented two different modes for the generation of patches. In the first mode, asynchronous patch generation, the response is returned to the user as soon as step 4 finishes. In this mode, the LUCID endpoint processes the arising data of the diff table at regular intervals (step 5 to 7). The other mode, synchronous patch generation, processes the steps 5 to 7 immediately after step 4, before returning the response to the user. Through this, the patch of an update operation will be available right after successful execution.

5.2 RDF Hashing Algorithm

We use an implementation of the algorithm described by [6] to create the hash value for a patch in triple based format.

To generate the same hash value for a patch regardless of which format, it is transformed to the triple based format before hashing as shown in Figure 3. We avoid the necessity of providing an URI schema for a patch by making it anonymous before hashing. That means, the subject URI of a patch and the URIs of intermediate resources and reified triples must be replaced by blank nodes before hashing. This allows for easier sharing of patches between participants in a distributed environment where usually each party uses their own scheme to create URIs. As one can attach arbitrary meta-information to the patch, the algorithm only considers a fixed set of properties for the hash calculation.

6. EVALUATION

Due to a lack of benchmarks for uniform and comparable testing of RDF-based versioning systems we now present three different use cases to evaluate the performance and correctness of our proposed VCS. We performed our evaluation on a machine with a 2 GHz Intel Core i7 quad-core processor and 16 GB of system memory. The used Virtuoso server version was 7.2.1 (07.20.3214-pthreads). It was configured to use 1 GB of available system memory.

6.1 Performance Benchmark

With the help of the Explore and Update use case of the Berlin SPARQL Benchmark (BSBM) [5], we measured the performance of our implemented VCS. The Explore use case consists of *SELECT*, *CONSTRUCT* and *DESCRIBE* SPARQL queries. The Update use case consists of *INSERT*

[8]https://www.w3.org/TR/rdf11-mt/#reification
[9]http://www.w3.org/ns/auth/cert#
[10]http://virtuoso.openlinksw.com/; minimum required version 7.2.1

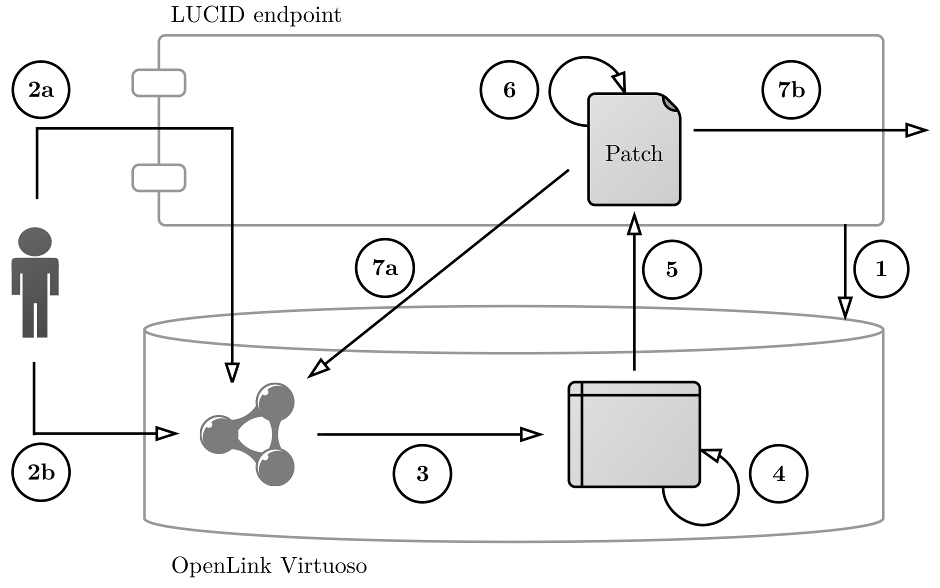

Figure 2: Architectural overview of our VCS implementation.

```
Input: RDF representation of patch
Output: hash value of patch
function calculateHash(patch)
    if quad based format then
        foreach revision do
            foreach delta do
                create intermediate resource;
                foreach triple do
                    create reified triple and attach to
                    intermediate;
                end
                replace delta named graph with
                intermediate;
            end
        end
    end
    replace URI of patch, intermediates and reified
    triples with blank nodes;
    calculate hash value of patch as described in [6];
    return hash value;
end
```

Figure 3: Calculation of the hash value of a patch.

DATA and *DELETE WHERE* SPARQL Update queries. We generated a benchmark dataset with around 10 million triples using the BSBM data generator[11]. Each benchmark run was performed with the BSBM test driver using 250 warm up and 500 query mix runs[12]. The results of the benchmark are shown in the chart of Figure 4. To have a

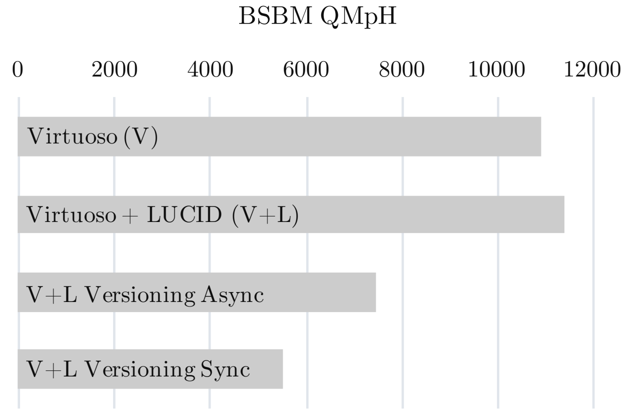

Figure 4: Comparison of Query Mixes per Hour (QMpH) of the Berlin SPARQL Benchmark (BSBM) Explore and Update use case.

reference value, we benchmarked a plain Virtuoso instance (V) that was able to perform 10909 Query Mixes per Hour (QMpH). To quantify the plain overhead of the LUCID endpoint, we run the test against an instance without versioning enabled using an equivalent Virtuoso instance as backend store (V+L). It achieved 11403 QMpH on average which is a 4.5% better performance than the plain Virtuoso instance (V)[13]. In the next test run we enabled versioning with asynchronous patch generation (*V+L Versioning Async*). The performance dropped by 35% to 7438 QMpH in comparison to the *V+L* run. Finally, we repeated the test with enabled versioning but this time using synchronous patch generation

[11] `./generate -pc 28482 -ud`

[12] `./testdriver SPARQL_ENDPOINT -runs 500 -w 250 -dg "urn:bsbm" -o run_001.xml -ucf usecases/exploreAndUpdate/sparql.txt -udataset dataset_update.nt -u UPDATE_ENDPOINT`

[13] There seems to be a problem related to the HTTP component of Virtuoso.

```
{
  ex:patch-af1a8a4b-... a eccrev:Commit ;
    eccrev:commitAuthor ex:MarvinFrommhold ;
    eccrev:commitMessage
      "extend property restriction of class a"
    prov:atTime
      "2015-12-17T13:37:00+01:00"^^xsd:dateTime ;
    eccrev:previousCommit ex:patch-8ca11472-... ;
    eccrev:previousCommitSha256
      "LNbDoZ...AFQ20="^^xsd:base64Binary ;
    eccrev:sha256
      "c7f9Hb...8Ft2E="^^xsd:base64Binary ;
    eccrev:hasRevision ex:revision-8686f2ab-... .
  ex:revision-8686f2ab-... a eccrev:Revision ;
    eccrev:hasRevisionGraph ex:GraphA ;
    eccrev:deltaDelete ex:delete-5391fd22-... ;
    eccrev:deltaInsert ex:insert-94c59669-... .
}
ex:delete-5391fd22-... {
  ex:ClassA rdfs:subClassOf [
    owl:onProperty ex:someProperty ;
    owl:cardinality "1"
  ] .
}
ex:insert-94c59669-... {
  ex:ClassA rdfs:subClassOf [
    owl:onProperty ex:someProperty ;
    owl:cardinality "1" ;
    owl:someValuesFrom ex:ClassC
  ] .
}
```

Listing 3: A patch in TriG syntax replacing an MSG due to the addition of triple `[] owl:someValuesFrom ex:ClassC` to the graph of the example dataset in Listing 1.

(*V+L Versioning Sync*). That means, the patches are created within the same transaction of the request right after the SPARQL Update query. This run provids feedback of the overall performance impact of our proposed VCS. The performance decreased to 5527 QMpH which is a drop of 51.5% compared to the *V+L* run.

A separate look at the average Query Execution Times (aQET) of the explore and update queries shows 78% slower update queries and 50% slower explore queries of the *V+L Versioning Async* run in comparison to the *V+L* run. On the other hand, the *V+L Versioning Sync* run has about 600% slower update queries, but nearly the same times for the explore queries, when compared to the *V+L* run.

In summary, the ideal configuration for the patch generation depends on the particular use case. If there is no need to have the patches available after an update immediately, the asynchronous patch generation is recommended due to the better overall performance. However, if the explore functionality of the system is of crucial importance, the synchronous patch generation should be used since only the update queries are affected by a performance hit.

6.2 Patch Size Benchmark

In addition to the overall performance test, we determined

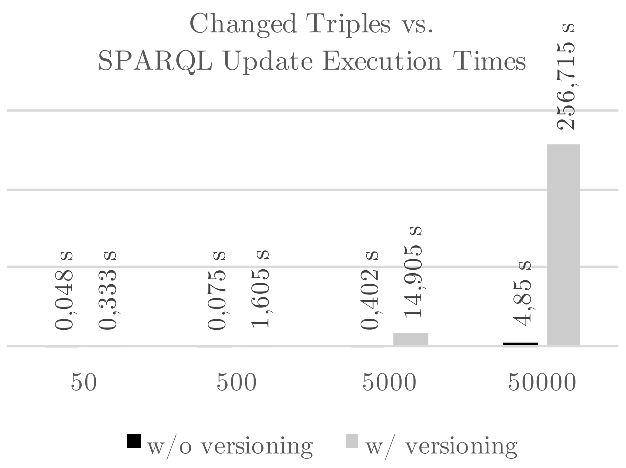

Figure 5: SPARQL Update query performance of the LUCID endpoint regarding the number of changed triples.

which size of updates (number of changed triples) our implementation is able to handle. For this, we created test datasets up to 50000 triples where each triple is of the form `<urn:s:i> <urn:p:i> <urn:o:i>` replacing i by 1 up to the specific size. We then executed a SPARQL Update query renaming the object of each triple. The results of a comparison between a LUCID endpoint with no versioning and a versioning enabled instance with synchronous patch generation is shown in the chart of Figure 5.

In a nutshell, SPARQL Update queries producing a large number of changed triples (> 10000) result in a clear drop in performance.

6.3 Versioning Evaluation Tool

To test the correctness of the patch generation of our system, we developed an evaluation tool[14]. It allows to provide a sample dataset description in TriG format as the starting point for a versioning operation and a SPARQL Update query which performs an alteration to the dataset. After submitting the data, the tool imports the given dataset, executes the SPARQL Update query and presents the resulting patch to the user allowing him to review the result.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a VCS for arbitrary RDF data. By supporting all concepts of the RDF specification, the VCS is able to track changes for any possible RDF dataset, even when blank nodes are involved. As provenance plays an important role in distributed environments, our work deals with the protection against unperceived history manipulation by providing hashing of patches. We described the concepts of our system and presented an implementation integrated into a middleware acting as a SPARQL proxy.

To provide a full-fledged versioning system, we plan to add more features in the context of the two research projects LUCID and LEDS. Most importantly, we want to add support

[14]A demo is available at https://versioning-evaluation-demo.eccenca.com.

for the basic operations described by Cassidy and Ballantine [7]. This allows, for example, to undo a change made mistakenly or merge different versions of a dataset enabling collaborative editing in distributed environments. As part of the LUCID project we are developing a Publish-Subscribe mechanism to synchronize patches between the participants of distributed B2B Linked Data networks. We will add signing for patches as trust plays an important role in this scenario. Currently, we only provide support for the OpenLink Virtuoso Universal Server. However, to ensure a wide application especially in enterprises, we are working on the support for additional RDF repositories, such as the Spatial and Graph option for Oracle Database 12c[15] and Complexible Stardog[16].

The evaluation highlighted that our current implementation is not able to handle large changes in a reasonable time. One of the main reasons causing this is the used implementation of the RDF hashing algorithm by [6]. The complete patch needs to be loaded into the memory to calculate its hash value. To resolve this issue a more scalable solution is required without the need to load the patch into the memory as a whole.

Finally, as more RDF versioning approaches arise, there is a need for a benchmark defining test scenarios for RDF versioning systems allowing to compare the performance of such systems.

8. ACKNOWLEDGMENTS

This work was partly supported by the following grants from the German Federal Ministry of Education and Research (BMBF) for the LUCID Project (GA no. 01IS14019A) as well as for the the LEDS Project (GA no. 03WKCG11A and GA no. 03WKCG11C).

9. REFERENCES

[1] L. Allen, G. Fernandez, K. Kane, D. Leblang, D. Minard, and J. Posner. Clearcase multisite: Supporting geographically-distributed software development. In *Software Configuration Management*, Lecture Notes in Computer Science, pages 194–214. Springer Berlin Heidelberg, 1995.

[2] S. Auer and H. Herre. A versioning and evolution framework for RDF knowledge bases. In *Perspectives of Systems Informatics*, Lecture Notes in Computer Science, pages 55–69. Springer Berlin Heidelberg, 2007.

[3] T. Berners-Lee and D. Connolly. Delta: an ontology for the distribution of differences between RDF graphs, 2004.

[4] C. Bizer and R. Cyganiak. RDF 1.1 TriG. *W3C recommendation*, 2014.

[5] C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *International Journal on Semantic Web & Information Systems*, 5(2):1–24, 2009.

[6] J. J. Carroll. Signing RDF graphs. In *The Semantic Web - ISWC 2003*, Lecture Notes in Computer Science, pages 369–384. Springer Berlin Heidelberg, 2003.

[7] S. Cassidy and J. Ballantine. Version control for RDF triple stores. *ICSOFT (ISDM/EHST/DC)*, 7:5–12, 2007.

[8] R. Cyganiak, A. Harth, and A. Hogan. N-quads: Extending n-triples with context. *W3C Recommendation*, 2008.

[9] R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 concepts and abstract syntax. *W3C Recommendation*, 2014.

[10] P. Gearon, A. Passant, and A. Polleres. SPARQL 1.1 update. *W3C recommendation*, 2013.

[11] Y. Gil, J. Cheney, P. Groth, O. Hartig, S. Miles, L. Moreau, P. P. da Silva, and Others. Provenance xg final report. *Final Incubator Group Report*, 2010.

[12] M. Graube, S. Hensel, and L. Urbas. R43ples: Revisions for triples. In *1st Workshop on Linked Data Quality*, 2014.

[13] A. Hogan. Skolemising blank nodes while preserving isomorphism. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 430–440. International World Wide Web Conferences Steering Committee, 2015.

[14] A. Kiryakov and D. Ognyanov. Tracking changes in RDF(S) repositories. In *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, Lecture Notes in Computer Science, pages 373–378. Springer Berlin Heidelberg, 2002.

[15] G. Klyne and J. J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. *W3C Recommendation*, 2004.

[16] T. Lebo, S. Sahoo, D. McGuinness, K. Belhajjame, J. Cheney, D. Corsar, D. Garijo, S. Soiland-Reyes, S. Zednik, and J. Zhao. Prov-o: The prov ontology. *W3C Recommendation*, 2013.

[17] A. Mallea, M. Arenas, A. Hogan, and A. Polleres. On blank nodes. In *The Semantic Web – ISWC 2011*, Lecture Notes in Computer Science, pages 421–437. Springer Berlin Heidelberg, 2011.

[18] N. Shadbolt, W. Hall, and T. Berners-Lee. The semantic web revisited. *Intelligent Systems, IEEE*, 21(3):96–101, 2006.

[19] G. Tummarello, C. Morbidoni, R. Bachmann-Gmür, and O. Erling. RDFSync: Efficient remote synchronization of RDF models. In *The Semantic Web*, Lecture Notes in Computer Science, pages 537–551. Springer Berlin Heidelberg, 2007.

[20] G. Tummarello, C. Morbidoni, P. Puliti, and F. Piazza. Signing individual fragments of an RDF graph. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 1020–1021. ACM, 2005.

[21] M. Vander Sande, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens, and R. Van de Walle. R&Wbase: git for triples. In *LDOW*. events.linkeddata.org, 2013.

[15] http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/rdfsemantic-graph-1902016.html
[16] http://stardog.com/