Open Semantic Revision Control with R43ples

Extending SPARQL to access revisions of Named Graphs

Markus Graube, Stephan Hensel, Leon Urbas
Chair for Process Control Systems Engineering
Technische Universität Dresden, Germany
{markus.graube, stephan.hensel, leon.urbas}@tu-dresden.de

ABSTRACT

The Semantic Web provides mechanisms to interlink data in a fast and efficient way and build complex information networks. However, one of the most important features missing for industrial application is version control which allows recording changes and rolling them back at any time if necessary. It is not sufficiently supported by today's triplestores and recent version control systems are not very well integrated into the Semantic Web. This paper shows a way of dealing with version and revision control using SPARQL. It presents R43ples as an approach using named graphs for semantically storing the differences between revisions. Furthermore it allows a direct access and manipulation of revisions with extended version of SPARQL. Smart mechanisms for restoring old revisions relying on query rewriting provide a fast way of querying old revisions of big datasets. A prototypical implementation of the system prove an appropriate performance under different conditions.

Keywords

Semantic Web, Revision Control, SPARQL, Named Graph, Query Rewriting

1. INTRODUCTION

The explosion of the Semantic Web in recent years [12] has provided the opportunity to develop advanced technology enablers to support new inter-organizational collaboration models towards the creation of virtual enterprises. Thus, the Semantic Web is going to become an important technology also in the manufacturing and process industries [5]. Cross-links to relevant collaboration partners are crucial key factors for companies in order to stay competitive. In fact, inter-enterprise collaboration through intense data sharing has become essential [13]. Especially small and medium-size enterprises have no other way to acquire a critical mass of resources and competencies. Communication and exchange of information are no longer instruments for process control but are key drivers of business performance, which can lead to the transformation of isolated individual companies towards an agile virtual enterprise.

The almost unlimited openness and flexibility of Linked Data can foster this transformation but has also some drawbacks. Industrial applications require reliability, security and stability in order to keep control of data manipulation. Thus, the application of semantic web technologies in industrial environments need further features. In fact according to the study of [16], version management is the most crucial requirement for software tools in the technical systems engineering process control. It is even more important than consistent and lossless data exchange as well as automated consistency check.

Almost every system is evolving due to changing requirements and system contexts. Due to the fact that systems usually interact with other systems following another life cycle, an important aspect is the trace of changes and the possibility to get back to a prior version when needed. Linked Data has a special demand for version control because of its very open nature and high amount of possible contributors to a data set. Through the application of version control, different contributors can independently apply changes to their data but there is always the possibility to merge into one common information base. Although there are different approaches for version management for the Semantic Web there are still open questions.

This paper presents the concepts and a prototypical implementation of the semantic revision control system R43ples (Revisions for Triples, pronounced as /'rɪpəls/) which has been briefly introduced in [4]. The focus of this paper are functionality and interfaces for revision control systems which is solved by a completely semantic approach for revisioning RDF data sets in named graphs. The data can be accessed via SPARQL as uniform Semantic Web interface with a few additional keywords. Thus, there is no need for a proprietary interface for data access and update. Furthermore it extends [4] by introducing a mechanism for querying old revisions based on query rewriting.

The remainder of the article is structured as follows. Section 2 provides a brief overview of the current state of art for version control in the Semantic Web. In section 3, the revision control concept of R43ples and the query rewriting mechanism are presented. Section 4 describes the prototypical implementation. Section 5 evaluates the implementation in terms of important metrics. Section 6 discusses the concept before the paper is concluded with an outlook of possible enhancements.

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

ISBN 978-1-4503-4752-5/16/09…$15.00

DOI: http://dx.doi.org/10.1145/2993318.2993336

2. RELATED WORK

There is a lot of previous work in versioning of models. For example, Watkins and Nicole [22] started with an ontology for modeling the provenance of documents defining a set of meta information for versioning. Taentzer et al. [18] distinguish between state-based and operation-based versioning systems which have different conflict detecting and handling mechanisms. Most models described in literature use entities with identifiers and do not rely on any order in a collection. Thus, the models can be easily handled as graphs fitting perfect in the world of the Semantic Web.

Different architectures for providing version control have been published [11], relying either on ontologies to represent changes, on additional information in databases or on facilitating an existing revision control system like Git. All have different ways to handle concurrent transactions. Some approaches for version control in the Semantic Web use locks to allow the simultaneously execution of independent transactions (e.g. [15]). Thus, all non-independent transactions will be blocked. The basic methodology for using locks was provided by Seidenberg et al. [17] to mitigate errors in a multi-user ontology editing environment.

Most authors that handle version control systems for the Semantic Web follow an operation-based approach which relies on specific operations and thus are not well integrated in current Semantic Web environment. Auer and Herre [1] build their concept on atomic changes on RDF graphs which are annotated in reified statements[1] of the original data. Another interesting vocabulary in this context is the change set schema from vocab.org[2]. It defines a possibility to annotate resources with their changes in time.

Cassidy and Ballantine [2] use context information in order to store information about patches. The changed triples are modeled as reified statements. Im et al. [8] use a delta-based approach for versioning RDF triples and introduce an *aggregated delta* concept which leverages the construction of version by storing additional deltas not only to the prior version but to all other versions. On the application side, the approaches for versioning are concentrating mainly on ontologies [21, 9] and do not care about instance data. Or they have features like synchronizing between different users [3] which is close to a version control system. However, this feature is deeply integrated into the specific application and its stack.

Vander Sande et al. [19] describe a concept for a version control system for triples. It defines add and delete sets in a quad store. Main disadvantage of the approach is that only parts of the system are modeled semantically. Other parts use for example hash tables to get relations between revisions and difference sets. Hauptmann et el. [7] propose a similar system with an internal cache mechanism in order to provide a high performance in terms of response time.

Another interesting approach which allows the tracking of information over time in Linked Data is the use of temporal RDF suggested by [6]. Here, triples do have a certain life cycle in which they are valid. We think that versioning compared to time labeling has the advantage that related changes are bundled in a semantic way and not only by having the same time stamp.

[1]http://www.w3.org/TR/rdf-mt/#Reif
[2]http://vocab.org/changeset/schema.html

3. CONCEPT

3.1 Overall Architecture

Version control can be differentiated into distributed systems and central systems. In a central system like Subversion[3] the whole repository is stored on a central server and the clients have local working copies. In distributed systems like Git[4], every client holds the full repository and can resynchronize with other clients. R43ples bases on a central repository because a client in a Semantic Web context can not checkout a complete local working copy. The revised graphs could be very huge because there are potential connections to other distributed information.

Consequently, pessimistic approaches following a conventional *Lock-Modify-Unlock* mechanism can not work in this context. Thus, an optimistic *Copy-Modify-Merge* mechanism) approach [14] is used which reflects the open nature of the Semantic Web better. Clients get their information via a SPARQL query (copy), change them in their local memory (modify) and afterwards commit their changes back merging them with other changes done in the meantime (merge).

R43ples works as a proxy in front of an existing triplestore. It modifies incoming queries by adding revision specific information and pass them to the attached triplestore. So, R43ples offers a slightly enhanced version of SPARQL. R43ples has no internal storage but saves all information in the attached triplestore.

R43ples handles version control on graph level and not on instance level. Thus, a specific version of a whole named graph is the unit under revision control. Unlike in file-based systems (as Subversion and Git) where a revision contains a set of files, a revision in R43ples contains only one single named graph. R43ples uses a triple as the smallest unit of manageable change. In conventional systems, the line ordering is used to localize changes (text-based) and to allow a delta based compression. Linked Data is graph-based and so current version control systems do not meet the localization mechanisms. Triples can be identified without a line context and thus be directly placed in add or delete sets for a revision. This makes it possible to implement a delta-based compression and avoid a pure binary-based storage of each revision.

3.2 Revision Model

R43ples can manage the revisions of multiple graphs at once. Therefore, R43ples uses the graph *http://eatld.et.tu-dresden.de/r43ples-revisions* to store all graphs under revision control together with a reference to a named graph containing the according revision information. Each of these revisions graphs has information about the complete history of the named graph with all change sets. The revisions and their connections are the main elements of the revision graph. They are described using the Revision Management Ontology (RMO)[5]. It extends the activities and entitities of PROV-O[6] which is used as foundation for provenance applications.

Figure 1 provides an examplary excerpt of a revision graph for *http://dbpedia.org/ontology* showing the two revisions

[3]http://subversion.apache.org/
[4]http://git-scm.com/
[5]https://github.com/plt-tud/r43ples/blob/master/doc/ontology/RMO.ttl
[6]http://www.w3.org/TR/prov-o/

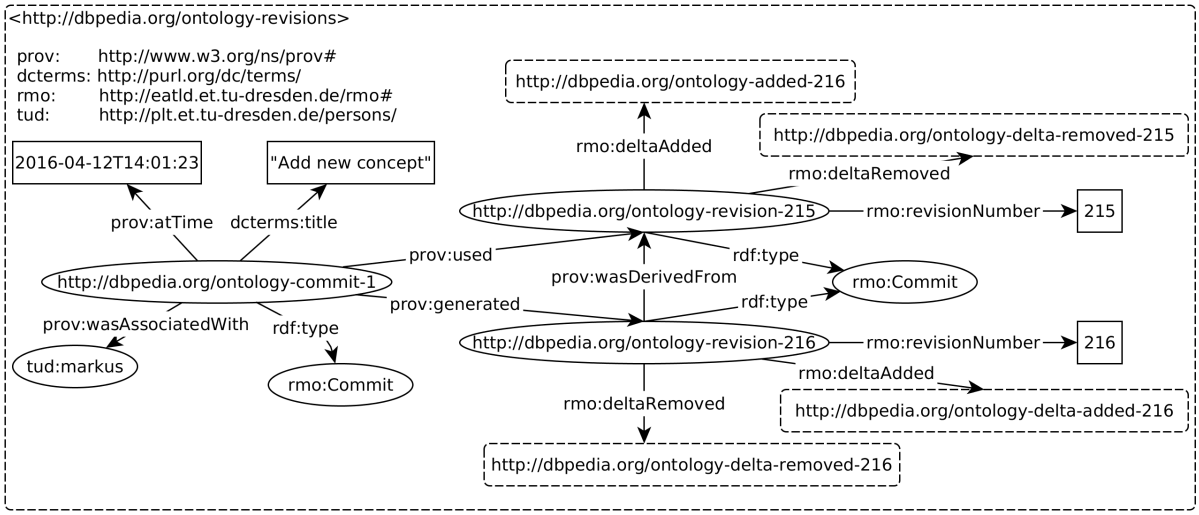

Figure 1: Excerpt of R43ples revision graph representing revision history of `http://dbpedia.org/ontology`

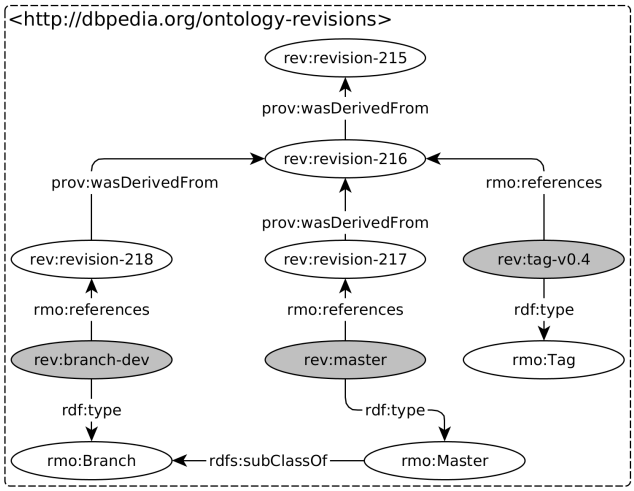

Figure 2: Modelling branches and tags (gray)

215 and 216. Both revisions are of type *rmo:Revision* (as subclass of *prov:Entity*) and contain a revision number for a simple human friendly representation (*rmo:revisionNumber*). The property *prov:wasDerivedFrom* connects consecutive revisions. The commit between two revisions is modeled as *rmo:Commit* (inherited by *prov:Activity*) connected via the attributed *prov:used* and *prov:generated*. It furthermore holds meta information about commit time (*prov:atTime*), commit message (*dcterms:title*) and the actor committing the change (*prov:wasAssociatedWith*). All differences between connected revisions are stored as change sets in further named graphs which are referenced by the properties *rmo:deltaAdded* and *rmo:deltaRemoved*.

Tags are references to specific revisions via the property *rmo:references* (as shown in Figure 2). They are of type *rmo:Tag* and have a unique name (*rmo:tagName*) as well as a description (*rdfs:description*). Similarly, branches are supported by allowing different successors of one revision. Each terminal revision of the generated branches is referenced by a *rmo:Branch* entity. The *rmo:Master* is a special subclass of *rmo:Branch* pointing to the default graph. Every branch and every tag has a fully materialized copy of the revision which is referenced by *rmo:fullGraph* property and can thus be queried directly without recreating data for this revision.

As the complete revision graph is just a named graph in the attached triple store, it can be accessed like any other data, for example via SPARQL queries.

3.3 SPARQL as interface for revision management

SPARQL is a quite prominent way to gather complex information within the Semantic Web. Thus, we have chosen it as base for querying and updating revisions of data sets. Besides, there exist different other approaches for dealing with Linked Data [20]. These are not part of this article.

Revision control requires that a specific revision can be specified for each named graph on which the query should be executed. This goal can be achieved by extending SPARQL in different ways. The following sections present different approaches. Each of them is illustrated with an example query which should query all friendships from revision 2 of the graph `http://test.com/r43ples-dataset1`.

3.3.1 Virtual Graph Interface

The first option is to use a virtual graph in the specification of the query. Here, R43ples interpretes everything after a fixed keyword (e.g. *revision/*) in the URI of the graph as identifier for a revision as shown in Listing 1. This option is easy to read and understand.

```
SELECT ?s ?o WHERE {
  GRAPH <http://test.com/r43ples-dataset1/
       revision/2> { ?s foaf:knows ?o. }
}
```

Listing 1: Query with Virtual Graph interface

3.3.2 Virtual Service Interface

The *SERVICE* functionality of SPARQL is the base for this approach. The query is performed on an unbound

graph. A virtual service *r43ples:revision* binds it to the desired revision of the graph by evaluating the semantic description of the revision and injecting the necessary content into the passed graph URI. As Listing 2 shows, the revision is specified semantically using the RMO vocabulary.

```
SELECT ?s ?o WHERE {
  GRAPH ?graph_rev2 { ?s foaf:knows ?o. }
  SERVICE <r43ples:revision> {
    ?graph_rev2 a rmo:Revision;
      rmo:revisionOf <http://test.com/r43ples-
          dataset1>;
      rmo:revisionNumber "2".
  }
}
```

Listing 2: Query with Virtual Service interface

3.3.3 Graph URI Function Interface

A filter function (*r43ples:revision*) for the graph URI can used similar to the virtual service interface. It allows to specify the revision number for the named graph which should be used for the query in this subgraph (as depicted in Listing 3).

```
SELECT ?s ?o WHERE {
  GRAPH r43ples:revision(<http://test.com/
          r43ples-dataset1>, 2)
    {?s foaf:knows ?o.}
}
```

Listing 3: Query with graph URI function interface

3.3.4 Additional Keyword Interface

The last concept includes an extension of the SPARQL language with additional keywords (similar to [10]). Here, a user can specify the revision identifier by using the keyword *REVISION* after specifying the graph (Listing 4).

```
SELECT ?s ?o WHERE {
  GRAPH <http://test.com/r43ples-dataset1>
      REVISION "2"
    {?s foaf:knows ?o.}
}
```

Listing 4: Query with *REVISION* keyword interface

3.3.5 Evaluation of the interfaces

All of the approaches have almost the same expressiveness and can be automatically transformed into each other. Of course, all approaches allow to specify a tag or branch name instead of a revision number and to perform a query on multiple graphs and multiple revisions. It is most likely that most of the time a human user should not come directly in touch with the SPARQL query. This should be handled by a user friendly application.

We decided to focus on the approach with the additional keyword *REVISION* since it makes the revision information explicit without chance for misinterpretation. For example, it's not clear in listing 1 if we want to retrieve information from revision 2 of *http://test.com/r43ples-dataset1* or just the content of the graph *http://test.com/r43ples-dataset1/revision/2* (which does not have any revision at all). Furthermore, other operations on revisions (updating, branching, taging, merging) is quite hard to solve without new keywords. The other options can also be implemented as further alternatives to specify a revision of a graph in parallel to the *REVISION* keyword approach,

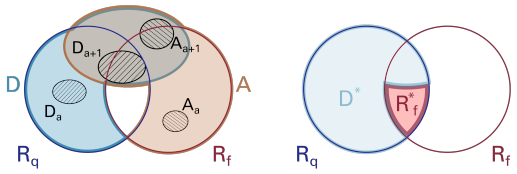

Figure 3: Triple sets of revisions R_q and R_f together with their changesets A and D (left) and together with sets D^* and R_f^* to retrieve revision R_q (right)

3.4 Querying Revisions

R43ples needs to roll back information of the named graphs to the revision specified by the client. Since all branches (respectively their leaf revision) or tags have a fully materialized copy of the referenced revision (*rmo:fullGraph*), the query on these revisions can directly be executed. For all other revisions, there are two different approaches.

3.4.1 Dynamic Generation of a Temporary Graph

The first approach is to generate a temporary graph which reconstructs all triples from the specified revision [4]. This requires getting a path from a revision with a full copy (tags or branches) to the specified revision. Then, the full graph is copied and all change sets along the revision path are applied to the copy. Afterwards the query is executed on the temporary graph, the results are passed to the client and finally the temporary graph is deleted.

3.4.2 Query Rewriting

The shortcoming of temporarily regenerating old revisions on demand is that the time to create this graph heavily depends on the size of the dataset [4]. Thus, it is not possible to generate a revision of a dataset with a few million triples in a user friendly time span. The alternative is to rewrite the query and execute all joins of change sets directly in the SPARQL engine of the attached triplestore (as e.g. proposed by [7]). Here, a rewriting engine parses the whole query. It iterates over all named graphs and searches for a path between the last revision of a branch and the specified revision in the query. Then, it rewrites each triple statement individually by embedding revision logic and the calculated revision path. Afterwards it reassembles the structure of the original query. Finally, the query is executed by a common SPARQL processor directly in the attached triple store.

Figure 3 shows the set concepts for retrieving an old revision of a graph from the current revision and the revision graph. We want to get set R_q containing the triples in revision q. The set R_f contains all triples of revision f which is referenced by a branch and thus has a fully materialized copy of this graph. Change sets have triples in an add set and in a delete set which are usually part of the relative complements of R_q or R_f (as depicted by A_a and D_a for a revision a with $q < a < f$). However, these can also be part of the union of R_q and R_f or even outside these sets, if they are first added, then deleted before added again (as shown by A_{a+1} and D_{a+1}). All change sets between both revisions f and q are stored in the delete set $D = \bigcup_{i=q+1}^{f} D_i$

and the add set $A = \bigcup_{i=q+1}^{f} A_i$. It is to be noted that parts of A and D can be outside of R_q and R_f if they are first added and afterwards deleted again. The task for restoring R_q from known R_f, A and D can be solved by joining R_f^* and D^* as shown in equation (1). R_f^* is the part of R_f which is not included in any add set and D^* are the parts of the delete sets that are not covered by earlier add sets (2). In order to ease the transfer to SPARQL, we treat the materialized graph as another delete set $D_{f+1} := R_f$ because both types can contain triples we are searching for and have to be combined in order to get the specified graph. This leads to equation (3).

$$R_q = R_f^* \cup D^* \quad (1)$$

$$R_q = \left(R_f \setminus \bigcup_{i=q+1}^{f} A_i\right) \cup \left(\bigcup_{j=q+1}^{f} \left(D_j \setminus \bigcup_{i=q+1}^{j} A_i\right)\right) \quad (2)$$

$$R_q = \bigcup_{j=q+1}^{f+1} \left(D_j \setminus \bigcup_{i=q+1}^{j} A_i\right) \quad (3)$$

In SPARQL, formula (3) has to be applied to each triple statement. It is searched in an unbound graph, which is determined by the union of the delete sets excluding the prior add sets. A simple example should illustrate this transfer. The query of Listing 4 (returning all connected ressources using the *foaf:knows* predicate) should be executed on revision $q = 2$ of graph *http://test.com/r43ples-dataset1* which has its *master* on revision $f = 5$. The rewritten query shown in Listing 5 searches for all occurences of the only statement in all graphs matching `?g_st1` (line 4). The rest of the query determines `?g_st1` which maps R_q into SPARQL.

First, lines 5–13 set `?g_st1` to a join of the full copy of revision 5 (`rev:5`) and the delete sets of the revisions between revision 5 and the successor of revision 2, namingly `rev:5`, `rev:4` and `rev:3`. Then, the occurences of the triple statements in add sets filter this result set (lines 14–21). Here, the query checks for every revision `?r_st1` from the first part if there exists an earlier (line 17) revision `?r_st1_add` which includes the matched triple statement. In this case this matching `?g_st1` is removed from the result set since the triple from the delete set of this revision does not contribute to the content of revision 2. However, the same triple can still be in the final result set if it also exists in an earlier delete set (and there is no further occurence in an even earlier add set).

Table 1 shows an examplary dataset on which the rewritten query is executed. It presents for each triple in every delete set if it is included in the graph `?g_st1` and thus not filtered out by the MINUS clause (line 14–21 from Listing 5). For the full graph there is only one triple that contributes to the final result set (`:Adam :knows :Emma`). The other triples are excluded since they are also matched in prior add sets. E.g. the triple `:Freddy :knows :Emma` is not included for the full graph of revision 5 since it also in the add set of revision 4. Although the triple `:Carlos :knows :Freddy` is not included for the full graph, it is included refering to the delete set of revision 4 and thus contributes to the final result set which is completed by triple `:Adam :knows :Bob` from revision 3.

The rewriting mechanism is not limited to simple queries but can handle multiple statements, multiple subgraphs as well as further SPARQL functions like filters and unions.

```
1  PREFIX rev: <http://test.com/r43ples-
       dataset1-revision->
2  SELECT DISTINCT ?s ?o
3  WHERE{
4    ## querying triple in unbound graph
5    GRAPH ?g_st1 { ?s foaf:knows ?o.}
6    ## joining delete sets and content of
         materialized branch (D_j)
7    GRAPH <http://test.com/r43ples-dataset1-
         revisions> {
8      { BIND(rev:5 AS ?r_st1)
9        [] rmo:references ?r_st1;
10          rmo:fullGraph ?g_st1.}
11     UNION {
12       ?r_st1 rmo:deltaRemoved ?g_st1 .
13       ## Revisions between revision q and f
14       FILTER(?r_st1 IN (rev:5,rev:4,rev:3))
15     }
16   }
17   ## removing content of add sets (A_i)
18   MINUS {
19     GRAPH ?g_st1_add { ?s foaf:knows ?o.}
20     GRAPH <http://test.com/r43ples-dataset1-
           revision> {
21       ?r_st1 prov:wasDerivedFrom* ?r_st1_add
             .
22       ?r_st1_add rmo:deltaAdded ?g_st1_add .
23       FILTER(?r_st1_add IN (rev:5,rev:4,rev
             :3))
24     }
25   }
26 }
```

Listing 5: Rewritten SPARQL query for applying the query from Listing 4 to revision 2

For example, Listing 6 presents a SPARQL query with a MINUS clause and two triple statements on a revised graph. The rewritten query in Listing 7 now contains for both triple statements the complex part of determing the delete sets and add sets which should be queried for this triple statement (analog to lines 7 to 25 in listing 5).

```
SELECT DISTINCT ?p1 ?p2
WHERE {
  GRAPH <http://test.com/r43ples-dataset1>
       REVISION "2" {
    ?p1 foaf:knows ?p2.
    MINUS {
      ?p1 foaf:knows :Danny.
    }
  }
}
```

Listing 6: SPARQL query with MINUS clause

3.5 Updating Revisions

R43ples supports the updating of a branch which is much easier than querying. The revision at the end of a branch already have a fully materialized copy of the graph. *INSERT* and *DELETE* queries are applied on the fully materialized graph as well as on newly generated change sets. Additionally, R43ples upates the revision graph by adding a *rmo:Commit* and *rmo:Revision* and moving the branch pointer. In order to fix issues with blank nodes, an skolem-

Table 1: Application of the rewritten query from listing 5 on an examplary dataset showing if the triples in the delete set are already included in prior add sets (`?g_st1`)

Revision	Graph	Content	?g_st1	
3	Add set	`:Adam :knows :Freddy`	–	
		`:Carlos :knows :Danny`	–	
	Delete set	`:Adam :knows :Bob`	✓	
4	Add set	`:Freddy :knows :Emma`	–	
	Delete set	`:Carlos :knows :Freddy`	✓	
5	Add set	`:Carlos :knows :Freddy`	–	
	Delete set	`:Carlos :knows :Danny`	✗	already included in add set of revision 3
master	Fully materialized Graph	`:Adam :knows :Emma`	✓	
		`:Adam :knows :Freddy`	✗	already included in add set of revision 2
		`:Carlos :knows :Freddy`	✗	already included in add set of revision 5
		`:Freddy :knows :Emma`	✗	already included in add set of revision 4

```
1  SELECT DISTINCT ?p1 ?p2
2  WHERE {
3    GRAPH ?g_st1 { ?p1 foaf:knows ?p2. }
4       # statements for determining ?g_st1
5       # (analog to lines 7-25 from listing 5)
6    MINUS {
7      GRAPH ?g_st2 { ?p1 foaf:knows :Danny. }
8        # statements for determining ?g_st2
9    }
10 }
```

Listing 7: Rewritten SPARQL query of Listing 6 including MINUS clause

```
MESSAGE "inserting sample triple"
INSERT DATA {
  GRAPH <http://test.com/r43ples-dataset1>
       REVISION "4"
  { <a1> <b1> <c1>. }
}
```

Listing 8: Updating revision 4 with R43ples update including a commit message using the MESSAGE keyword

ization step is necessary before updating the graphs.

The interface follows the SPARQL 1.1 Update pattern. It specifies the revision information using the keyword *REVISION* (analog to the query method) and a commit message by the keyword *MESSAGE* as shown in Listing 8. The committer should be extracted from an authorization framework. If the specified revision does not refer to a branch the server will deny the update query.

Every revision can also be tagged for an easy identification using the keyword *TAG*. In a similar way also branches can be created with the keyword *BRANCH*. Thus, also commits on old revision are possible if a respective branch has been created.

If a system support branches, it also requires a possibility to merge two branches afterwards. This should be supported by the new SPARQL command *MERGE*. The detailed functionality of a good semantic merging interface is part of further research.

4. IMPLEMENTATION

A prototype of R43ples was implemented as a Java application. Jersey[7] is used as RESTful Web service framework and grizzly[8] as the web server. Jena ARQ library handles all SPARQL queries and their result handling. R43ples is only loosely coupled to the attached triplestore by providing different extendable access mechanisms (TDB, JDBC, HTTP interface) to the triplestores. R43ples is provided as open source and can be retrieved from Github[9]. A demonstration system is running on http://eatld.et.tu-dresden.de/r43ples/sparql.

We implemented both concepts for querying revisions (temporary graph and query rewriting) which can be switched by the HTTP parameter *join_option*. R43ples provides a debug interface allowing the execution of SPARQL queries directly on the attached endpoint without managing revision information. Currently, no authentication is included. Thus, the committing user is specified with the further keyword *USER* in the query. All important parameters can be configured during startup (named graph for storing revision information, server URL and port for providing service, HTTPS key, attached triplestore). For the sake of self-description, R43ples copies the SPARQL 1.1 Service Description[10] of the attached endpoint and adds *sd:r43ples* as further *sd:Feature* in order to show its extended capabilities.

5. EVALUATION

Important metrics for evaluating revision control systems are query performance as well as storage costs. Looking at the latter one, R43ples requires an amount S_{Revision} of additional triples for a new revision i which is almost proportional to the size of changes ($S^i_{\text{Changes}} = S^i_{\text{Add}} + S^i_{\text{Del}}$) and independent from the complexity of previous revisions or the revision graph (4). The additional triples in the revision graph (six for the revision node; six for the commit node) are negligible. Creating a branch (5) or tagging a revision i (6) stores a copy of the full graph (S^i_{Graph}) in addition to a fixed number of triples.

[7] https://jersey.java.net/
[8] https://grizzly.java.net/
[9] https://github.com/plt-tud/r43ples
[10] http://www.w3.org/TR/sparql11-service-description/

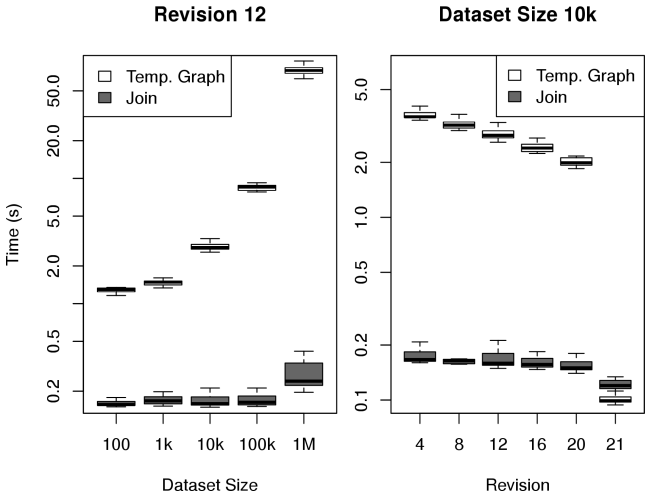

Figure 4: Comparison of R43ples response times between query rewriting option (gray) and temporary graph option (white) for increasing dataset size (left) and different revisions (right)

$$S^i_{\text{Revision}} = S^i_{\text{Changes}} + 11 \quad (4)$$

$$S^i_{\text{Branch}} = S^i_{\text{Graph}} + 13 \quad (5)$$

$$S^i_{\text{Tag}} = S^i_{\text{Graph}} + 13 \quad (6)$$

The perfomance while querying the last revision in a branch is equal to the situation without R43ples since these graphs are completey materialized. For older revisions, we have measured the response time for *SELECT* queries. Thus, we have set up different configurations varying the size of the data set and the revisions path lengt since these have been the most influent factors in [4]. We generated random data sets with sizes of 100, 1k, 10k, 100k and 1M triples without blank nodes. Then, we created 21 revisions for each data set with changes of 100 triples in each revision. Finally, a client executed a simple SPARQL query (querying all attributes for a specific resource). The time between sending the request and receiving the response was measured. This procedure was repeated 20 times for each configuration to capture random effects such as computing load or garbage collection. We evaluated the operation time of R43ples on a 4 GB RAM system running a Stardog 4.0[11] as SPARQL endpoint connected to R43ples.

Figure 4 shows that the approach with the temporary graph is heavily dependent both on the size of the dataset (left subfigure with increasing dataset size and fixed revision 12) and on the distance of the queried revision to a branch (right subfigure with decreasing size of revision path length and fixed dataset size). The drop in the response time between revision 20 and 21 is due to the fact that revision 21 is the branch revision with a fully materialized graph and can be queried directly. For earlier revisions the whole dataset has to be copied which takes about two seconds for this size of dataset. Then the application of the change sets is almost linear to the number and size of the change sets.

The query rewriting option is less dependent on both factors. For this query and dataset, it is almost as fast as a direct query on the appropriate revisions. However, the type of query in combination with the attached triplestore has a strong influence on the performance. There are even cases (querying all triples) which can require more time than the temporary graph approach.

In order to show the feasibility for real world datasets, we conducted another evaluation with a dump of 130 DBpedia Live changesets[12]. The changesets are quite big with over 10.000 triples each and summarize to a dataset of 3.7 million triples. This size makes the temporary graph approach unusable at all. Afterwards, we executed a simple query[13] on the last revision using the graph rewriting option. The results show an average response time of 6.5 seconds for the master revision. For every revision going down to the past this time increases about 10 seconds. Due to the fact that the delete sets of every revisions have to be compared against every prior add set, the increase is more than linear.

6. DISCUSSION

The approach presented here promotes a deep semantic description of the revision information. This allows other clients to reconstruct revisions and opens the approach for additional algorithms. Clients can use the endpoint offered by R43ples with or without using the extended revision control functionality. The interface is completely transparent to SPARQL clients. SPARQL queries without additional revision information work on the master revision. However, it is easy for clients to check the endpoint's capabilities by querying its service description.

Most of the time, clients will work on *MASTER* or other branches since usually the most recent information is needed. This makes it more important to raise the performance of accessing these revisions than saving storage. Hence, the storage of these terminal revisions seems to be an appropriate solution for balancing performance and storage.

As long as a client queries revisions referenced by branches or tags, the query performance is equal to the situation without a revision control system. For all other revisions, R43ples offers two algorithms. The on-the-fly generation based on generating a temporary graph is stable and sufficient for small to medium sized data sets. It has advantages, if the query needs to access nearly all triples inside the graph. However, increasing sizes of datasets requires a change in accessing old revisions to a rewriting of the query. Here, the SPARQL join option is better for bigger datasets since there is no need to copy the dataset. However, also this type of query can be very time consuming due to the arbitrary path length operator (line 17 in Listing 5) and the undetermined order of execution of the subqueries. Furthermore, the performance is strongly dependant on the SPARQL query approach in the attached triplestore. An automatic decision under which circumstances which approach should be used is a goal of future work as well as integrating other more advanced reconstructing mechanisms. Such an alternative could be the use of internal cache for determining if a triple in a change set should be queried as suggested by Hauptmann et al. [7].

Both query algorithms currently use a very simple ap-

[11] http://stardog.com/

[12] http://live.dbpedia.org/changesets/2015/06/03/

[13] SELECT * WHERE { GRAPH <http://dbpedia.org> REVISION "x" {?s dbpedia:contributor ?o. } }

proach for determining a revision path from a branch to the specified revision. A shortest-path-solver could also find other branches or tagged revisions that are closer to the revision which can result in a higher performance. The complexity of the approach is necessary in order to reflect the possility to add and delete the same triple multiple times.

The integration of R43ples into existing software tools for e.g ontology engineering or authoring requires that these tools work on a triplestore with SPARQL interface. Under these circumstances it would be minimal effort to exchange the SPARQL interface with the slightly enhanced R43ples interface.

7. CONCLUSIONS

This paper provides a concept and prototypical implementation of an open semantic revision control system. It adds revision information in semantic models and uses the capabilities of SPARQL for accessing and modifying revisions. It is possible to retrieve an old revision in a reasonable amount of time while working on recent revisions without any loss in performance. The advantage of our approach is that it is completely based on semantics and stored in a system independent way in any attached triplestore, which makes it easy to apply the concept in various areas. It should facilitate the development of other algorithms for reconstructing old revisions. The slight enhancements of new keywords should be considered as a starting point for discussing to extend SPARQL for version control.

The presented concept needs further research in the area of performance and additional functionality. Our next steps will involve an intensive consideration of how this concept can be integrated into existing tools and how securtiy mechanisms can be integrated.

8. ACKNOWLEDGMENTS

The research leading to these results was partially funded by the European Community's Seventh Framework Programme under grant agreement no. FP7-284928 ComVantage.

9. REFERENCES

[1] S. Auer and H. Herre. A versioning and evolution framework for RDF knowledge bases. In *Perspectives of Systems Informatics*, pages 55–69. Springer, 2007.

[2] S. Cassidy and J. Ballantine. Version control for RDF triple stores. *ICSOFT*, 7:5–12, 2007.

[3] T. Ermilov, N. Heino, S. Tramp, and S. Auer. Ontowiki mobile–knowledge management in your pocket. In *The Semantic Web: Research and Applications*, pages 185–199. Springer, 2011.

[4] M. Graube, S. Hensel, and L. Urbas. R43ples: Revisions for triples - an approach for version control in the semantic web. In M. Knuth, D. Kontokostas, and H. Sack, editors, *Workshop on Linked Data Quality (LDQ)*, 2014.

[5] M. Graube, J. Pfeffer, J. Ziegler, and L. Urbas. Linked data as integrating technology for industrial data. *Internation Journal of Distributed Systems and Technologies*, 3(3):40–52, 2012.

[6] C. Gutierrez, C. Hurtado, and A. Vaisman. Introducing time into RDF. *IEEE Transactions on Knowledge and Data Engineering*, 19(2):207–218, Feb. 2007.

[7] C. Hauptmann, M. Brocco, and W. Wörndl. Scalable Semantic Version Control for Linked Data Management. In *Workshop on Linked Data Quality (LDQ)*, 2015.

[8] D.-H. Im, S.-W. Lee, and H.-J. Kim. A version management framework for RDF triple stores. *International Journal of Software Engineering and Knowledge Engineering*, 22(01):85–106, Feb. 2012.

[9] M. Luczak-Rösch, G. Coskun, A. Paschke, M. Rothe, and R. Tolksdorf. SVoNt-version control of OWL ontologies on the concept level. 176:79–84.

[10] M. Meimaris, G. Papastefanatos, S. Viglas, Y. Stavrakas, C. Pateritsas, and I. Anagnostopoulos. A query language for multi-version data web archives.

[11] R. Mordinyi, E. Serral, D. Winkler, and S. Biffl. Evaluating Software Architectures using Ontologies for Storing and Versioning of Engineering Data in Heterogeneous Systems Engineering Environments. In *Proc. of ETFA*, 2014.

[12] J. Murdock, C. Buckner, and C. Allen. Containing the semantic explosion. In *Proc. of PhiloWeb*, Lyon, 2012.

[13] T. Münch, R. Buchmann, J. Pfeffer, P. Ortiz, C. Christl, J. Hladik, J. Ziegler, O. Lazaro, D. Karagiannis, and L. Urbas. An innovative virtual enterprise approach to agile micro and SME-based collaboration networks. 2013.

[14] T. Redmond, M. Smith, N. Drummond, and T. Tudorache. Managing Change: An Ontology Version Control System. In *OWLED*, 2008.

[15] S. Scheglmann, S. Staab, M. Thimm, and G. Gröner. Locking for Concurrent Transactions on Ontologies. In *The Semantic Web: Semantics and Big Data*, number 7882 in LNCS, pages 94–108. Springer, 2013.

[16] N. Schmidt, A. Lüder, H. Steiniger, and S. Biffl. Analyzing requirements on software tools according to the functional engineering phase in the technical systems engineering process. In *Proc. of ETFA*, 2014.

[17] J. Seidenberg and A. Rector. A Methodology for Asynchronous Multi-user Editing of Semantic Web Ontologies. In *Proc. of the 4th Int. Conf. on Knowledge Capture*, pages 127–134. ACM, 2007.

[18] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software & Systems Modeling*, pages 1–34, 2012.

[19] M. Vander Sande, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens, and R. Van de Walle. R&wbase: Git for triples. In *Proceedings of the 6th Workshop on Linked Data on the Web*, 2013.

[20] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert. Triple pattern fragments: A low-cost knowledge graph interface for the web. 37:184–206.

[21] M. Völkel and T. Groza. SemVersion: An RDF-based ontology versioning system. In *Proceedings of the IADIS international conference WWW/Internet*, volume 2006, pages 195–202. Citeseer, 2006.

[22] E. R. Watkins and D. A. Nicole. Version control in online software repositories. In *Int. Conf. on Software Engineering Research and Practice*, volume 2, pages 550–556, 2005.