R43ples: Revisions for Triples

An Approach for Version Control in the Semantic Web

Markus Graube
Chair for Process Control Systems Engineering
Technische Universität Dresden, Germany
markus.graube@tu-dresden.de

Stephan Hensel
Chair for Process Control Systems Engineering,
Technische Universität Dresden, Germany
stephan.hensel@tu-dresden.de

Leon Urbas
Chair for Process Control Systems Engineering,
Technische Universität Dresden, Germany
leon.urbas@tu-dresden.de

ABSTRACT

For most use cases, the Semantic Web provides essential mechanisms to interlink data in a fast and efficient way. However, it is still not widely accepted in industry since some important features are not mature enough. Requirements include easier model transformation and access to dynamic data. One of the most missing important features is version control which would make it possible to record changes in a way that they can be rolled back at any time. Recent version control system are not very well integrated into the Semantic Web.

This paper shows a novel way of dealing with version control for Linked Data. It presents R43ples as an approach using named graphs to semantically store the differences between revisions. Furthermore it allows direct access and manipulation of revisions with SPARQL. Thus, the access is almost transparent for the clients which can still use known SPARQL queries enhanced with some additional keywords. A prototypical implementation of the system shows a proof of concept and performance considerations.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query languages*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Information networks*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*

General Terms

Linked Data, Versioning, SPARQL, Revision, Query, Named Graphs

1. INTRODUCTION

The explosion of the Semantic Web in recent years [9] has provided the opportunity to develop advanced technology enablers to support new inter-organisational collaboration models for the creation of virtual enterprises. The ComVantage[1] project explores the capabilities of Linked Data (LD) as a flexible and rapidly unifying way to provide access to the data vaults of all stakeholders of a virtual enterprise by creating a product-centred collaboration space. However, the almost unlimited openness and flexibility of Linked Data may also involve disadvantages. Industrial applications require reliability, security and stability. Thus, they need to keep control over the process of data manipulation. In fact, version control is an essential requirement for them to adapt this technology.

Section 2 of this paper states the need for version control systems in the Semantic Web and provides an overview of related work and our contributions. In section 3, the version control concept of R43ples is presented. Section 4 describes the prototypical implementation. Section 5 evaluates the concept and gives some metrics of the implementation. Section 6 discusses the concept further before the paper is concluded with an outlook of possible enhancements.

2. BACKGROUND

2.1 Linked Data

Linked Data is a set of best practices for modelling and interconnecting information in a widely accepted semantic way. It is becoming more and more important in the world of Linked Open Data. It uses the Resource Description Framework (RDF) as the base model. RDF handles information as a semantic network of single statements consisting of subject, predicate and object. LD information entities are referenced by URIs. A named graph is a collection of RDF statements grouped together and identified by a URI. Named graphs are a kind of transformation of quads (a triple with a fourth element). SPARQL (SPARQL Protocol And RDF Query Language) is the dominant query language in the Semantic Web. It uses a graph-based matching mechanism with powerful filter and aggregating functionality and additional support for named graphs. Nevertheless, Linked Data might also be a useful technology for industrial environments [6]. This requires controlled write mechanisms to the Linked Data cloud as stated by Berners-Lee and O'Hara [2]. Version control could be one way to achieve such a controlled read-write mechanism.

[1]EU FP7 Integrated Project "Collaborative Manufacturing Network for Competitive Advantage": www.comvantage.eu

LDQ 2014, 1st Workshop on Linked Data Quality Sept. 2, 2014, Leipzig, Germany.

2.2 Version Control for Triples

The major function of version control systems is to record changes in the information model in order to get back to a prior version when needed. Furthermore, version control makes it possible to merge changes of different authors into one common information base. Obviously, this functionality is not only needed for software engineering but for data in general. This includes Linked Data which has a special demand for version control because of its very open nature and the number of possible contributors to a data set.

Current version control systems are usually either text-based (changes can be localised in lines) or completely binary (no localisation of changes possible). However, Linked Data is graph-based and thus in this case the existing systems don't meet the localisation mechanisms which is necessary for merging revisions. Additionally, one can differentiate between distributed systems and central systems. In a central system like Subversion[2] the whole repository is stored on a central server and the clients have local working copies. In distributed systems like Git[3], every client holds the full repository and can re-synchronise with other clients.

2.3 Related Work

2.3.1 Model Versioning

There has been a lot of previous work on versioning of models. For example, Watkins and Nicole [16] started with an ontology for modelling the provenance of documents defining a set of meta information for versioning. Taentzer et al. [12] distinguish between state-based and operation based versioning systems which have different mechanisms for conflict detecting and handling. However, although versioning of models is a key technique in model driven engineering, it is not supported by a widely accepted concept. Most models described in the literature use entities with identifiers and don't rely on any order in a collection. Thus, they can be easily handled as graphs, which fits the base model in the Semantic Web.

2.3.2 Temporal RDF

Another interesting approach which allows tracking of information over time in Linked Data is the use of temporal RDF suggested by [7]. However, we think that versioninghas the advantage over time labelling that related changes are bundled in semantic way and not only by the same time stamp. Furthermore, there is no query for temporal RDF available that has a good compatibility with SPARQL.

2.3.3 Semantic Web Versioning

Most authors who handle version control systems for the Semantic Web follow an operation based approach which relies on specific operations and are thus not well integrated in the current Semantic Web environment. Auer and Herre [1] base their concept on atomic changes to RDF graphs which are annotated in reified statements[4] of the original data. The approach of Cassidy and Ballantine [3] uses context information in order to store information about patches. The changed triples are on the other hand modelled as reified statements. Im et al. [8] use a delta-based approach for versioning RDF triples and introduce an *aggregated delta* approach which leverages the construction of a version by storing additional deltas not only to the prior version but to all other versions.

Some Semantic Web applications support synchronising between different users, e.g. OntoWiki Mobile [4]. This is close to a version control system. However, this feature is deeply integrated into the specific application and its stack.

The concept of Vander Sande et. al. [13], based on [14], for version control seems to meet almost all requirements for the Semantic Web. Unfortunately, only parts of the system are modelled semantically, e.g. other parts may use hash tables to get relations between revisions and difference sets. Furthermore, the distributed nature of Git is not utilised despite of the promising title of the article.

2.4 Contributions

R43ples offers a completely semantic approach for versioning RDF data sets in named graphs and accessing them via SPARQL. The concept is based partly on the work of Vander Sande et. al. [13]. However, our approach has no need for additional languages since we use the SPARQL 1.1 features for updating data. This can be done with adding a few keywords to SPARQL. Furthermore, we propose a model of revision information describing both commits as well as changes in a purely semantic way using named graphs instead of additional look-up tables. Finally, we provide a performance evaluation of a prototypical implementation.

3. CONCEPT

3.1 Graph Based Version Control

We use a central repository since no local working copy in a traditional sense can be checked out in the Semantic Web and held on the client. The complete graph could be extremly large and every piece of information is potentially connected with other information spread over the global Linked Data cloud. This also excludes conventional Lock-Modify-Unlock mechanisms. This would imply that the whole network has to be locked. Thus we use a Copy-Modify-Merge mechanism where clients get their information via SPARQL (copy), work with this in their local memory (modify) and commit their updates to the server via SPARQL again (merge). This makes it possible for users to keep on working with the well-known SPARQL interface while providing fast and flexible revisions management.

R43ples handles version control on a graph level and not the instance level. Thus, a specific version of a whole named graph is the unit under version control. It is stored as a revision which can be queried and used as a base for further changes. Unlike in file-based systems (e.g. Subversion or Git) where a revision contains a set of files representing a specific point in time, a revision in R43ples contains only one single named graph.

3.2 Semantic Revision Model

3.2.1 Data Model of Revisions

The whole approach uses semantics in order to avoid hidden meanings which makes it hard for other clients to access the

[2] http://subversion.apache.org/
[3] http://git-scm.com/
[4] http://www.w3.org/TR/rdf-mt/#Reif

information. Thus, revisions are modelled as Linked Data. The data model uses PROV-O [15] as base ontology and is extended by some attributes. The vocabulary is called Revision Management Ontology (RMO). Figure 1 shows an excerpt of a graph revision model, with one commit generating a new revision for a specific named graph (marked in grey). The revisions are linked to the named graph *http://test* (via the property *rmo:revisionOf*) and contain a revision number (*rmo:revisionNumber*) for a simple human friendly representation. The property *prov:wasDerivedFrom* connects two revisions and describes the revision graph. The commit between two revisions is modelled as standard *prov:Activity* connected via *prov:used* and *prov:generated* attributes. It holds meta information about commit time (*prov:atTime*), commit message (*dcterms:title*) and the actor committing the changes (*prov:wasAssociatedWith*).

3.2.2 Naming Graphs for Storing Revisions

The named graph with the URI of the revisioned graph holds the *MASTER* revision representing the terminal revision of the default branch in the revision graph. The information about other revisions and their connections and further revisioned graphs is stored in an additional named graph for each revisioned graph called *<r43ples-revisions>*. All revision control systems have to provide information of all revisions while handling the number of storage. Since "97,3% of the entire data in each version remains unchanged" [8] it is necessary to compress this data. Delta-based storage is the approach of choice here. According to [10] RDF triples are the smallest unit of change and are thus the basis for calculating the differences as deltas between revisions. The differences of revisions are again a set of triples and can be stored in additional named graphs. Every revision consists of one *ADD* set and one *DELETE* set assigned with the properties *rmo:deltaAdded* and *rmo:deltaRemoved*. Applying these delta sets to the prior revision will lead to the current revision.

3.2.3 Tags and Branches

The R43ples approach supports tags as 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, different branches are supported by allowing different successors of one revision via *prov:isDerivedFrom*. Each terminal revision of the generated branches is referenced by a *rmo:Branch* entity. The *rmo:Master* is a subclass pointing to the default graph. All these references point to copies of a full graph of this revision via *rmo:fullGraph* property.

The centralised approach of R43ples can easily achieve the necessary uniqueness of the revision numbers. The revision numbers can follow different schemes, for example just ordinals or using a hash. We decided for a more complex naming scheme which indicates the position of a revision in the graph. For the system these are just strings for providing a human-friendly identifier without semantic meaning (although the revision number, not shown in figure 2, is also part of the URI, e.g. "3.1-22"). The users need to be able to retrieve the whole revision graph including the numbers of the revisions. With R43ples it is possible to receive this information like any other data via SPARQL queries directly on the revision graph *<r43ples-revisions>*.

3.3 Dynamic Handling of Revisions

3.3.1 Querying Revisions

Information from the *MASTER* revision is instantly available since the whole data set exists in the specified named graph. It is used when the client does not specify a revision. Therefore, it is likely that it will be accessed very often.

However, other revisions must be generated dynamically as only the delta information is stored between two revisions. With respect to the revision to be generated, all triples of the *add* set must be added to the the previous revision and all triples of the *delete* set must be removed from the previous revision. R43ples accepts slightly enhanced SPARQL queries which allow to add the revision number for each specified graph in the SPARQL query. For each named graph g specified in a query, a temporary graph TG_g, r is generated for the specified revision r according to equation 1 (g_x = full materialised revision x of graph g):

$$TG_{g,r} = g_{\text{nearestBranch}} + \sum_{\text{revision } i=r}^{\text{nearestBranch}} (deleteSet_{g,i} - addSet_{g,i}) \tag{1}$$

This simple formula can be mapped to a series of SPARQL queries as presented in the pseudo code below. It firsts creates a graph *<graph-rev_g>* merging all change sets. Afterwards it rewrites the query so it uses this new temporary graph instead of the specified one. The result of the SPARQL query on that graph is returned after cleaning up the temporary graph.

```
def select_query(query):
    for (graph,rev_g) in query.graphs_and_revs():
        sparql("COPY GRAPH <graph> \
                TO GRAPH <graph-rev_g>")
        for rev in graph.path_to_revision(rev_g):
            sparql("REMOVE GRAPH <rev.add_set_graph>
                    FROM GRAPH <graph-rev_g>")
            sparql("ADD GRAPH <rev.delete_set_graph>
                    TO GRAPH <graph-rev_g>")
        query.replace(graph, "graph-rev_g")
    result = sparql(query_string)
    for (graph,rev_g) in query.graphs_and_revs():
        sparql("DROP GRAPH <graph-rev_g>")
    return result
```

When considering the merging of revisions, it does not matter which previous revision is used to generate the merged revision due to the properties of SPARQL. An *INSERT* statement of an existing triple does not insert it a second time and a *DELETE* statement of a non-existing triple does not end in an error message. The *add* set A and *delete* set D of a revision with the set of triples R_m merged from revision with sets of triples R_1 and R_2 must comply with the rules from equations 2 and 3.

$$A = (R_m \backslash R_1) \cup (R_m \backslash R_2) \tag{2}$$

$$D = (R_1 \backslash R_m) \cup (R_2 \backslash R_m) \tag{3}$$

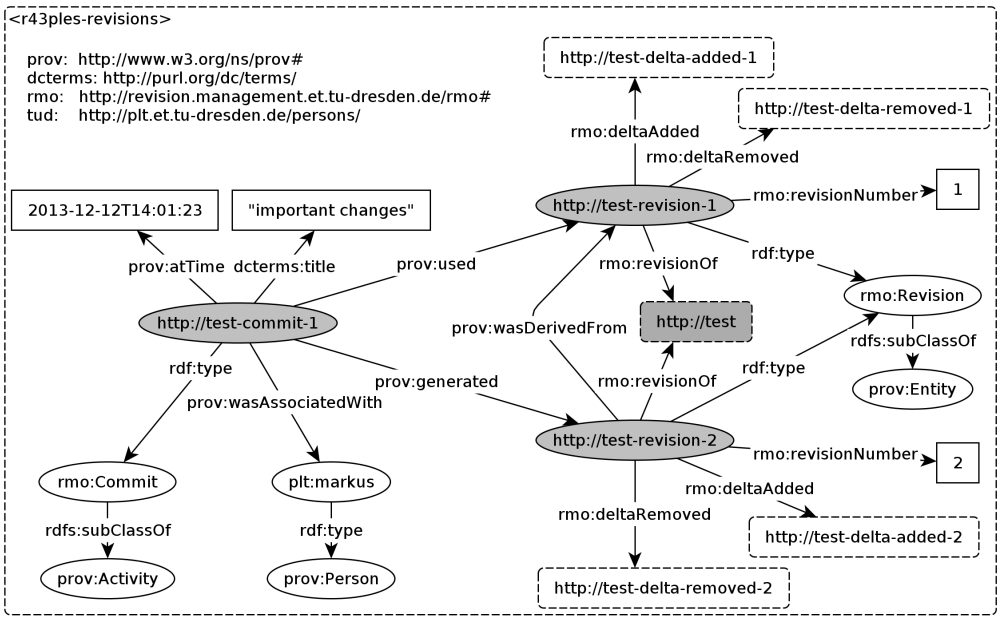

Figure 1: Data Model of a revision graph with ontology RMO

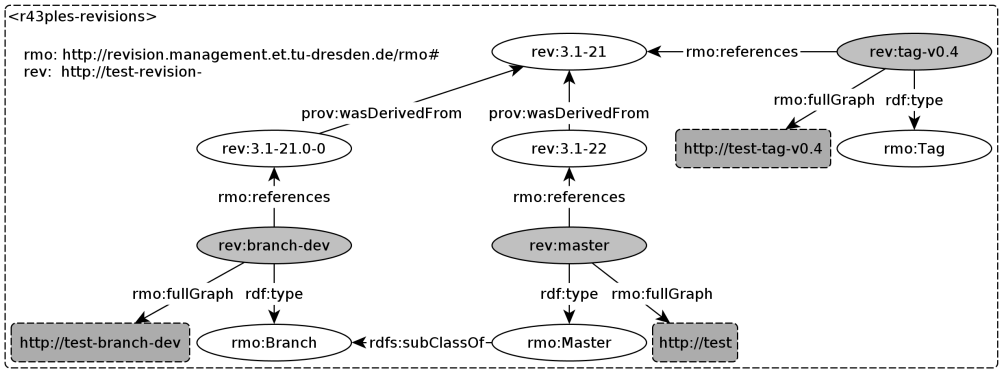

Figure 2: Model of master, branches and tags

3.3.2 Updating Revisions

Clients update revisions via the established SPARQL UPDATE command. This updates the revision graph with a new revision node which references the new change sets. The changes are both reflected in the new add and delete sets as well as in the updated full graph. However, updates can only be performed on the terminal sibling of a branch.

If a client wants to update a revision which is not referenced by a branch, the commit is rejected. The client has to merge its local changes with the most recent information of the branch. Merging is the application of two different change sets to one entity. If the local merge is possible, the client can recommit these merged changes. The other option is to explicitly create a new branch for the local changes.

The client cannot usually merge if it is unable to reconcile the changes. These conflicts have to be resolved afterwards in order to get a common consolidated data model in the revision control system. Thus, the changes have to be combined or one change has to be selected in preference to the other. This is performed via an additional administrator interface on the server.

3.4 SPARQL extension for R43ples

In a SPARQL query it has to be possible to determine the revision of the involved named graphs. Furthermore, update queries should contain information about the author and a commit message. Partly, this information could be embedded into the name of the graph. However, we strongly believe that loading identifiers with semantics would be a violation of the basic principles of Linked Data. Another option are new keywords or specifying this information as part of the WHERE clause as triple patterns like `?revision rmo:revisionOf <sampleGraph> ; rmo:revisionNumber "43"`. However, the latter one has the disadvantage that there is no clear distinction between the specification of revision information and SPARQL query pattern.

We decided to introduce the additional keyword *REVISION*

```
SELECT ?s ?p ?o
FROM <sampleGraph> REVISION "43"
WHERE {
  ?s ?p ?o.
}
```

Listing 1: SELECT query for revision 43 of *graph*

```
USER <mgraube>
INSERT DATA INTO <sampleGraph> REVISION "
    MASTER" MESSAGE "Small change"
{ <a> <b> <c>. }
```

Listing 2: Update query building on top of revison 42

to SPARQL to add the necessary semantic. Furthermore, the update mechanisms need some meta information about the commit introduced by the keywords *USER* and *MESSAGE*. Finally, the creation of tags and branches is solved by the keywords *BRANCH* and *TAG*.

In a *SELECT* query the user can define the revision number by applying the *FROM* clause with the keyword *REVISION*. It can be a number representing a revision, a string representing a branch or tag (e.g. "master") or empty. When it is empty or the keyword *REVISION* is missing, the *MASTER* revision will be used as default. An exemplary query is shown in listing 1.

Updates (*INSERT* or *DELETE* queries) can only be executed on a branch specified by the branch name or the number of a revision referenced by a branch. In *INSERT* and *DELETE* queries the performing user must first be defined. Therefore the keyword *USER* is reserved. After the FROM respectively the INTO clause the keyword *REVISION* identifies the graph revision following the same approach as in a *SELECT* query. Furthermore, there could be attached a commit message following the keyword *MESSAGE* as shown in listing 2.

The *REVISION* parameter is necessary for the SPARQL endpoint to check to which branch revision a client wants to apply its changes. If the client wants to update a revision that is not directly referenced by a branch, the server will reject the commit. Then, the client needs to check if its data model is consistent with the new information from a branch revision. If so, it can resubmit its changes, or it can open a new branch if there is a conflict the client is not able to handle. If the branch revision of the server matches that of the client, the server will accept the change and create a new revision with the information provided. Then, the responding branch reference will be forwarded to this new revision.

Listing 3 depicts a SPARQL query for generating a new branch. In the example, a new branch is created with the information from revision 42. The same interface is available for creating a tag using the keyword *TAG* instead of *BRANCH*.

```
USER <mgraube>
BRANCH <sampleGraph> REVISION "42" TO "
    Feature xyz"
```

Listing 3: Query for branching from revision 42

HTTP Parameter	Description
graph-revision-number	Revision of *graph* of last query
graph-revision-number-of-master	Current *MASTER* revision number of *graph*

Table 1: HTTP header parameters

The clients are kept aware of the recent *MASTER* revision in every SPARQL response. The HTTP response header is extended by additional fields which specify the current *MASTER* revision number and the revision number on which the query was executed for every named graph involved. Table 1 describes the construction of the parameter names. All underlined sub strings are replaced with the current named graph under version control. This information is not needed by the client for querying. Yet it provides the new revision number after a commit and is thus very useful for the client.

4. IMPLEMENTATION

The concept was implemented as proof of concept and its source code is publicly available via GitHub[5]. The prototype is realised as a SPARQL proxy rather than a modification of an existing open-source SPARQL endpoint. The implementation works as a Java application. Jersey[6] is used as RESTful (Representational State Transfer) Web service framework and grizzly[7] as the web server while Virtuoso[8] acts as triple store and SPARQL endpoint. A live demonstration system is running on http://eatld.et.tu-dresden.de:8890/r43ples/sparql.

Figure 3 shows the system structure. If a client wants to use the revision control features of R43ples it has to send the SPARQL queries to R43ples' SPARQL endpoint instead of the triplestore's endpoint. Furthermore there is an administrator interface which acts as a test bed for functions that don't yet have a proper REST interface. These functions are controlled by a command line interface and perform complex management of the graphs under version control.

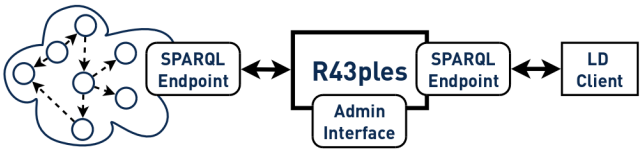

Figure 3: System Structure

R43ples stores no information about the revisions itself but

[5] https://github.com/plt-tud/r43ples
[6] https://jersey.java.net/
[7] https://grizzly.java.net/
[8] http://virtuoso.openlinksw.com/

```
CONSTRUCT {?s ?p ?o} WHERE {
  GRAPH <NEW-REV-TEMP> { ?s ?p ?o }
  FILTER NOT EXISTS { GRAPH <LAST-REV> { ?s
      ?p ?o } }
}
```

Listing 4: Get all added triples

uses a configured triplestore which is accessed by the triple store interface. The communication is based on SPARQL queries. To ensure the integrity of the data, only the SPARQL proxy should have access to the different graphs which it creates. Access rights are defined in the triple store. The clients need to know if the endpoint supports R43ples features in addition to standard SPARQL. Hence, R43ples copies the SPARQL 1.1 Service Description[9] of the connected endpoint and adds *sd:r43ples* as further *sd:Feature*.

The implemented proxy SPARQL endpoint can also handle standard SPARQL queries. Of course, this raises the requirement that the revisioned graph shall be only edited by R43ples and its specific queries. Otherwise inconsistencies would be generated. Virtuoso supports such access policies for the SPARQL endpoint, prohibiting write access to the *<r43ples-revisions>* graph and all graphs which are related to R43ples.

The generation and update of the version system information is completely implemented with the help of SPARQL queries. R43ples performs a SPARQL update on a temporary copy of the full graph of the specified branch. Afterwards, it retrieves all added triples with the SPARQL query from listing 4 which returns all triples which are in *NEW-REV-TEMP* but not in *LAST-REV*. After the same concept was used for the removed triples, the *ADD* and *DELETE* sets are constructed with the help of a SPARQL *CONSTRUCT* query. Then the new revision information is inserted in *<r43ples-revisions>* and the actual full graph is updated with the help of *INSERT* and *DELETE* queries.

The administrator interface offers an additional way for interacting with R43ples for those features which don't have a friendly REST interface yet. Those tasks are currently:

- Put an existing graph under revision management
- Import a new graph under version control
- Generate visualisation of the revision graph (yEd export)
- Set a new *MASTER* revision
- Merge two revisions

The admin interface currently supports turtle serialisation[10] for the export and import of RDF data. The visualisation of the revisions, their connections and branches is done by creating a GraphML file which can be viewed with yEd[11].

[9] http://www.w3.org/TR/sparql11-service-description/
[10] http://www.w3.org/2007/02/turtle/primer/
[11] http://www.yworks.com/de/products_yed_about.html

The merging feature is still under construction while we are investigating different approaches for a user friendly interface.

5. EVALUATION

5.1 Response Time

An important metric for evaluating the usability of this concept is the response time of the service for R43ples queries in various configurations. Therefore, we have measured the time between the request sent by the client and the response received using Apache jMeter[12]. We evaluated the operation time of R43ples in a complex setup on a 4 GB RAM system running a Virtuoso 7 as SPARQL endpoint connected to R43ples. We generated random data sets with sizes of 100, 1000, 10000 and 100000 triples. Then we created ten revisions for each data set with changes of 10 to 100 triples. Finally, we measured the response time for a simple SPARQL query (querying all triples and limiting them to ten results) dependent on all data sets, all revisions and all different change sizes. The measurement was repeated 20 times to capture random effects such as computing load.

Figure 4 presents some results showing the response time in comparison to variations of the three variables around a specific setup (1000 triples in the data set, going back five commits into the past with 50 triples changing in every commit). The left plot shows that the response time increases linearly with the number of commits plus a constant bias of some milliseconds. The size of the commit seems also to be almost linear to the response time as suggested by the middle plot. Even the size of the data set has linear influence (note the logarithmic scale in the right plot).

A deeper analysis shows that the structure of the data set is not significant. The overhead for querying a revision that is available as full graph is about 10 ms in comparison to a direct SPARQL query and is thus almost negligible. However, if the revision has to be generated by R43ples, the dominant factors are the overall size of changes to be reversed and the size of the data set. Equation 4 lists a simple linear model which almost exactly reflects these findings ($R^2 = 0.98$) with the variables T as R43ples response time in milliseconds, S_{DS} as data set size, S_C as change size and P as path length to a full graph revision. Thus, in many application T would be of order $\mathcal{O}(S_{DS})$.

$$T = 100 + 0.06 * S_{DS} + 0.7 * (P * S_C) \quad (4)$$

The results makes sense since the algorithm has to duplicate the graph and then apply all changes. Both efforts are proportional to the size. As minor result R43ples can perform few revisions and big changes in each revision step better than lots of small changes assuming that the overall number of changed triples is the same. Furthermore, UPDATE query time increases linearly with the size of the committed change set.

5.2 Storage

The costs for a new revision $S_{\Delta,\text{Revision}}$ (in additional triples) are almost proportional to the size of changes and independent from the complexity of previous revisions and the revision graph ($S_{\Delta,\text{Revision}} = S_C + 12$). The additional fixed

[12] http://jmeter.apache.org/

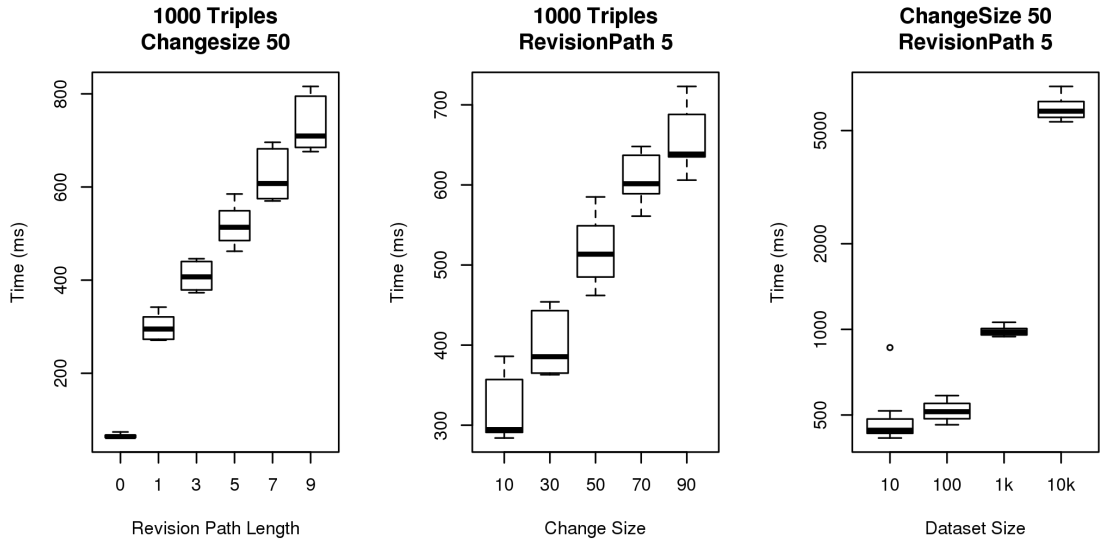

Figure 4: R43ples response time in comparison to the revision path length (left), the change size of the single commits (middle), and the size of the data set (right)

triples in *<r43ples-revisions>* (six for a revision; six for the commit) are negligible. The creation of a branch or a tag copies the full graph besides the addition of a fixed number of triples ($S_{\Delta,\text{Tag}} = S_{DS} + 11$; $S_{\Delta,\text{Branch}} = S_{DS} + 15$).

6. DISCUSSION

Although the approach presented here solves most of the versioning problems, there are also some drawbacks.

Named graphs are used extensively, mainly for storing differences between revisions. This means that the use of named graphs for other purposes cannot be guaranteed. Those purposes could be structuring of information, access control or additional provenance information. One might ask if we need an additional context attribute as a fifth element explicitly declared for revision control.

The concept is fully transparent for SPARQL clients which are not aware of the R43ples version control system. They can use the prototype as common SPARQL endpoint without additional features always working on the master revision. Clients can easily check if an endpoint supports R43ples query by evaluating the service description of the endpoint.

Clients will usually work on *MASTER* or other branches in order to get the most recent information. However, there could be situations when clients should continuously work on a specific revision of a graph. Then, this revision of the graph should be tagged in order to store a full copy. A possible solution would be the automatic detection of such frequently used revisions and triggering of tag generation.

Another drawback is the lack of support for blank nodes in the current implementation. You can't assume that blank nodes from different graphs with the same blank node identifiers are the same. For example, the blank nodes in the change sets are not equal to the ones in the full graph, prohibiting a correct application of the changes when generating an old revision. This can of course be solved by a prior Skolemization which should ideally be performed by the client or could also be achieved by an enhanced version of R43ples before executing a SPARQL query.

Currently, the generation of uncached revision follows a simple approach applying all changes from the first successor until reaching the leaf of a branch. However, if there are many tags in the revision graph, it could be more efficient to use another revision path to generate this revision. Thus, one has to solve a shortest-path-problem.

Another point of discussion is the way of transferring the necessary additional information. Currently, the R43ples SPARQL server transports the *MASTER* revision as well as the relevant revisions of all involved graphs in the HTTP header. On the other hand, the R43ples clients transport information about the graph revisions in the HTTP body. An alternative would be to transfer both information in the HTTP body and thus on the same level. This would need an extension of the SPARQL result model.

The integration of version control into the existing Semantic Web tool environment is not easy. A basic requirement is that these tools don't work on a file basis but on a triplestore with SPARQL interface. Under these circumstances it would be no big problem to exchange the SPARQL interface with the slightly enhanced R43ples interface.

The performance of the prototype limits the application to medium sized data sets. Queries on data sets with more than a few thousand triples take longer than most users are willing to wait. This can be solved by splitting large data sets into smaller ones and by directly implementing the concept into the SPARQL endpoint which should improve

performance considerably. Another promising approach we are currently working on is the use of enhanced SPARQL rewriting in order to perform the query taking into account the full graph and all change sets in one request Hence, the generation of the whole graph for the specified revision is not necessary which really takes long time for big datasets.

Finally, security is a crucial point for all industrial applications. We rely on the adaptable security mechanisms of existing triple stores and SPARQL endpoints. These should only provide information about the revision tree and the revisioned data sets to authenticated and authorised users. This could be achieved for example by the approach suggested by [11, 5].

7. CONCLUSIONS

We have presented a concept for a semantic revision control system for Linked Data which uses the capabilities of SPARQL. The implemented prototype works well for querying cached graphs. The generation of uncached graphs is sufficient for small to medium sized data sets. The advantage of our approach is that it is completely based on semantics and thus the information about revisions can be retrieved via SPARQL. Furthermore, SPARQL is used as access mechanism with only slight adaptations in order to ensure the semantic use of revision information while keeping the query compatible to standard SPARQL.

However, this concept still needs further research. Our next steps will involve investigating different merging approaches and an intensive consideration of how this concept can be integrated into existing tools.

8. ACKNOWLEDGEMENTS

This research was partly funded by the European Commission on the grant number 284928 (ComVantage).

9. REFERENCES

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

[2] T. Berners-Lee and K. O'Hara. The read-write linked data web. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1987):20120513–20120513, Feb. 2013.

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

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

[5] M. Graube, P. Ortiz, M. Carnerero, O. Lazaro, M. Uriarte, and L. Urbas. Flexibility vs. security in linked enterprise data access control graphs. In *Proc. of 9th IEEE Int. Conf. on Information Assurance and Security*, 2013.

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

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

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

[10] D. Ognyanov and A. Kiryakov. Tracking changes in RDF(S) repositories. In *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, page 373–378. Springer, 2002.

[11] P. Ortiz, O. Lazaro, M. Uriarte, and M. Carnerero. Enhanced multi-domain access-control for secure mobile collaboration through linked data cloud in manufacturing. In *Proceedings of IEEE World of Wireless Mobile and Multimedia Networks (WoWMoM) conference 2013*, 2013.

[12] 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*, page 1–34, 2012.

[13] 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.

[14] 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.

[15] W3C. PROV-O: the PROV ontology, Apr. 2013.

[16] E. R. Watkins and D. A. Nicole. Version control in online software repositories. In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice*, volume 2, page 550–556, 2005.