R&Wbase: Git for triples

Miel Vander Sande
miel.vandersande@ugent.be

Pieter Colpaert
pieter.colpaert@ugent.be

Ruben Verborgh
ruben.verborgh@ugent.be

Sam Coppens
sam.coppens@ugent.be

Erik Mannens
erik.mannens@ugent.be

Rik Van de Walle
rik.vandewalle@ugent.be

Ghent University - iMinds
Department of Electronics and Information Systems, Multimedia Lab
Gaston Crommenlaan 8 bus 201
B-9050 Ledeberg-Ghent, Belgium

ABSTRACT

Read/Write infrastructures are often predicted to be the next big challenge for Linked Data. In the domains of Open Data and cultural heritage, this is already an urgent need. They require the exchange of partial graphs, personalised views on data and a need for trust. A strong versioning model supported by provenance is therefore crucial. However, current triple stores handle storage rather naïvely and don not seem up for the challenge.

In this paper, we introduce R&Wbase, a new approach build on the principles of distributed version control. Triples are stored in a *quad*-store as consecutive deltas, reducing the amount of stored triples drastically. We demonstrate an efficient technique for storing different deltas in a single graph, allowing simple resolving of different versions and separate access. Furthermore, provenance tracking is included at operation level, since each commit, storing a delta and its metadata, is described directly as provenance. The use of branching is supported, providing flexible custom views on the data. Finally, we provide a straightforward way for querying different versions through SPARQL, by using virtual graphs.

1. INTRODUCTION

In the Linked Data world, Read/Write infrastructures are often predicted to be the next challenge [3], both in a large-scale context such as the Web and in smaller contexts like data publishing or harvesting. This challenge is currently discussed in several research areas. First, there is *big data management*. The use of RDF allows a rapid integration of many data sets, resulting in a fast increase in volume. Furthermore, when many different parties have write access, content grows fast (*e.g.*, social media). Second, with data originating from many different sources, the notion of *trust* becomes vital. Users will require a personal view on data, according to their trusted sources. Versioning data, and tracking, managing and tracking provenance is therefore a crucial part. Third, in the domains of *Cultural Heritage and Linked Open Data*, concrete issues have already been specified.

In the field of Cultural Heritage, harvesting metadata is a common practice. The large amounts of exchanged metadata require harvesting techniques to be cost-effective. The widely-used HTTP- and XML-based protocol OAI-PMH [1] allows incremental harvesting, which only collects the latest changes. This reduces network traffic and other resource usage. Recently, cultural heritage has more and more realized the benefits of machine-readable RDF [12]. Clearly, these evolutions reflect strongly on triple storage, which has a huge impact on today's stores (*e.g.*, OpenLink Virtuoso, Sesame, and AllegroGraph). Concerning the challenges ahead, current solutions store triples rather naïvely, which result in inefficient storage. Different versions of data can be stored in different graphs, but this leads to a duplication of all triples. Furthermore, managing or accessing changes is not supported natively, which makes incremental harvesting a painful task.

Another important domain is Linked Open Data. Current initiatives involve a unidirectional system where consumers download datasets from data portals, published by governments. However, with the key goal of public participation in mind, the Open Data movement is moving towards Open Data Ecosystems [8]. This introduces Open Data life cycles by installing a feedback loop. Feedback allows, for instance, consumers to patch data in order to improve data quality. Beyond doubt, strong version management will be crucial, together with automatic provenance tracking and publishing. Data providers and consumers need custom views on data, only based on modifications from trusted parties.

In this paper, we introduce R&Wbase (Read-and-Write base), a combined solution for meeting the requirements described above. We propose a new approach for reading and writing triples by applying distributed version control principles. We include provenance tracking at operation level, and publish it with every query result.

This paper is structured as follows. We start by discussing some related work in Section 2. Next, Section 3 outlines our approach for distributed triple version control in triple stores. The feasibility is then demonstrated with a Proof of Concept in Section 4. Finally, we discuss some future work in Section 5 and end with a conclusion in Section 6.

2. RELATED WORK

There has already been significant work on distributed version control in triple stores. One solution is SemVersion [13], a CVS[1]-based RDFS and OWL versioning system. It manages versioned models, *i.e.*, ontologies under versioning, which hold references to different versions. A version is a snapshot of the graph and holds

[1]Concurrent Versions System - *http://www.nongnu.org/cvs/*

different metadata, including a reference to the parent version and a link to its provenance. Related, more complex work discusses version control in the context of replicating partial RDF graphs [9]. This approach aims at lightweight clients (*e.g.*, mobile devices) that cannot store many triples or handle expensive processes like difference calculation or conflict resolving. *Triple bitmaps* are used to encode which triples are included in the partial graph and which are not. These bitmaps are used after the modifications to merge the partial graph back into the full graph. Cassidy *et al.* propose an approach based on Darcs' *theory of patches* [2], wherein a version if described as a sequence of patches. Each patch uses named graphs to identify a graph with deletions and one with additions. There is only one version, where all patches are applied sequentially.

The previously mentioned works all succeed in providing versioning for RDF. However, they do not all reduce the storage overhead. Also, the way these versions are accessed is not clear, nor can they be queried separately. Branching and parallel versions are only supported in SemVersion, but cannot be resolved at query-time, since they involve heavy recalculations. Provenance can perhaps be linked to a specific version, but is not automatically available using the tracked changes. Finally, the formalisation of deltas (or patches) is either unknown, or causes a lot of overhead.

Im *et al.* discuss a version management framework [7], which uses one original version and deltas to reduce storage space. Different versions are constructed at query-time and use delta compression to increase performance. These versions can exists in parallel, can be merged and form different branches. A custom relational database is used to store triples. A version is constructed by using SQL queries. Although this results in high performance, it is relational database dependent. This causes interoperability issues with existing triple stores and prevents migration to more optimal storage solutions.

A very popular approach to version control for source code of software is *Git*. Git is a fast, scalable, distributed revision control system with an unusually rich command set that provides both high-level operations and full access to internals [11]. Although it is not perfect for datasets, Git has already been used as the engine behind a couple of Open Data Ecosystems [8]. For instance, the city of Montpellier[2] and the city of Chicago[3] created a Github account. They have published their data in XLS, CSV, KML, SHP, and so on. The metadata are stored in README files: who is the owner, when was it published, category, description, license, and so on. In Chicago, not only did they release the data, they also included examples on how to use the data with R, a functional programming language to statistically explore datasets. Stefan Wehrmeyer created a Git repository[4] to track changes in the German lawbook. Taking this initiative as an example, The Netherlands[5] and Flanders[6] have done the same. To publish Linked Data using Git, RDF triples need to be serialized into a certain format first. In 2011, this was done by Ross Singer for the MARC codes, a US standard for bibliographic catalogues: he specified a normalized *Turtle* format in the README of his repository and published the description of all the codes in separate Turtle files[7].

[2]*https://github.com/VilleDeMontpellier/*

[3]*https://github.com/Chicago/*

[4]*https://github.com/bundestag/gesetze*

[5]*https://github.com/statengeneraal/wetten-tools*

[6]*https://github.com/okfn-be/codex-vlaanderen*

[7]*https://github.com/rsinger/LinkedMARCCodes/*

3. DISTRIBUTED TRIPLE VERSION CONTROL

Git is a great tool for text-based files, such as source code, but it is not perfect for data. In this section, we give an overview of our approach, which assembles the right version of the data at query-time, for applying distributed version control principles to a triple store.

3.1 Storage structure and algorithm

Our approach is to store only the *deltas* between each version of a triple set, instead of storing each version in its entirety. This reduces the amount of stored triples and will result into more efficient storage. A delta consists of a set of *additions* and a set of *deletions*; modifications to a triple are considered a deletion immediately followed by an addition. To achieve this, all triples are stored internally as *quads* [4], consisting of the original triple and a *context value*, identifying the version and indicating whether the triple was added or deleted. We thus provide an extra interpretation layer above a traditional quad store, providing an interface to a versioned triple store.

For every delta, we need to distinguish between the *add-set* and the *delete-set*, and each sets need to correspond to exactly one delta. Therefore, we identify every new delta with an even number $2y$ that is larger than all preceding delta numbers. All triples that have been added in this version compared to the previous are assigned a context value of $2y$. All triples that have been removed in the version are assigned a context value of $2y+1$. Furthermore, the delta identifier $2y$ is used to store the delta's metadata in triple format, such as indicating the delta's parents, the date, the person responsible for the changes, *etc.* This collection of metadata is known as a *commit* and is discussed in Section 3.2. The initial delta is the first one added to the triple store, and will always have an empty delete-set.

The interpretation layer is responsible for translating SPARQL queries such that the underlying quad store appears as a regular triple store. To find the answer to a query on version $2y_n$, we first find all its ancestors $A_{2y_n} = \{2y_i, \dots, 2y_j\}$. This is done by traversing the metadata for $2y_n$ and can be cached to speed up subsequent requests. An extra constraint is added to the query that the triples should correspond to a quad whose context value is either $2y_a$ or $2y_a+1$ ($2y_a \in A_{2y_n}$), *e.g.*, triples that belong to either the add-set or delete-set of versions from which $2y_n$ derives. From this result set, we only need the triple with the highest context value. This can result in three possible cases for each triple (as shown in Figure 1):

the triple is not returned: the triple was not part of any version and therefore *may not* be part of the result;

the triple has an even context value: the most recent change of the triple in the version's ancestry was an addition and therefore *must* be part of the result;

the triple has an uneven context value: the most recent change of the triple in the version's ancestry was a deletion and therefore *may not* be part of the result.

This indicates how the presence of a context element allows us to store a history of additions and deletions with only deltas, while allowing an efficient access pattern to the data because of carefully crafted version identifiers.

3.2 Commits and Branching

Following the principles of Git, changes are made using *commits*. A commit is a delta and associated metadata, consisting of a unique identifier, a message, a reference to the parent commit, a reference to the author of the changes, and a reference to the committer. In a Semantic Web context, this collection of metadata is equivalent

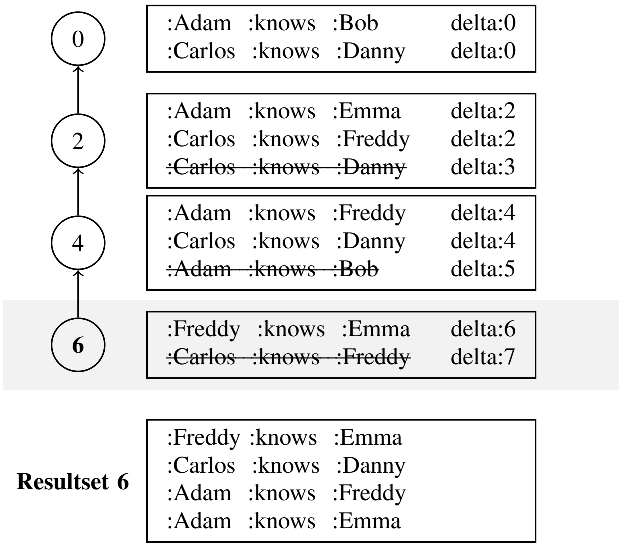
0
:Adam :knows :Bob delta:0
:Carlos :knows :Danny delta:0
2
:Adam :knows :Emma delta:2
:Carlos :knows :Freddy delta:2
~~:Carlos :knows :Danny~~ delta:3
4
:Adam :knows :Freddy delta:4
:Carlos :knows :Danny delta:4
~~:Adam :knows :Bob~~ delta:5
6
:Freddy :knows :Emma delta:6
~~:Carlos :knows :Freddy~~ delta:7
Resultset 6
:Freddy :knows :Emma
:Carlos :knows :Danny
:Adam :knows :Freddy
:Adam :knows :Emma

Figure 1: In every version, even deltas indicate additions and uneven deltas indicate deletions. The total result-set of 6 thus only includes triples of which the highest context value is even.

to the notion of provenance, as defined by the W3C Provenance Incubator Group [6]. Therefore, each commit can be formalized in RDF using the PROV-O vocabulary, as as demonstrated in Listing 1. For this approach, the context value refers to a certain delta, and user information is gathered from the authentication system of the triple store. The identifier is an externally unique hash, derived from the combined commit information. This hash is used to exchange commits between different systems, enabling distributed techniques similar to *push* and *pull* in Git. Internally, a lookup table maps each identifier to a corresponding delta identifier, so the paths can still be properly resolved.

```
@prefix prov:   <http://www.w3.org/ns/prov#>.
@prefix dcterms:  <http://purl.org/dc/terms/>.
@prefix ex:  <http://example.com/vocab#>.
@prefix version: <http://example.com/graphs/versions/>.
@prefix commit: <http://example.com/commits/>.
@prefix : <http://example.com/persons/>.

commit:hIjKlMn a prov:Activity;
   prov:atTime      "2013-02-16T01:52:02Z";
   prov:used        version:aBcDeFg;
   prov:generated  version:hIjKlMn;
   dcterms:title "Update social graph.";
   prov:wasAssociatedWith :Derek .

version:aBcDeFg a prov:Entity, ex:Dataset .

version:hIjKlMn a prov:Entity, ex:Dataset;
   prov:wasDerivedFrom version:aBcDeFg .

:Derek a prov:Person .
```

Listing 1: Derivations can be directly descibed as provenance using PROV-O

In the previous section, we mentioned the paths between the different deltas. In practice, commits will provide the structure of these paths through their parent relation. Because each commit only refers to its predecessor, commits can be structured in a very flexible way. This enables the use of *branches*. This implies that paths can run in parallel, creating a connected, directed, acyclic graph, where paths divert from and merge with each other (as shown in Figure 2).

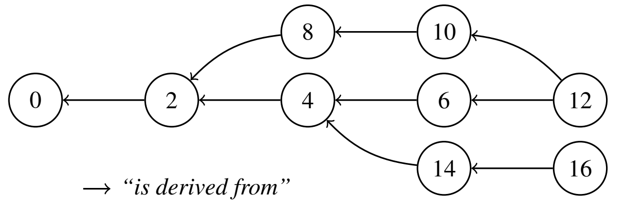

Figure 2: A triple store is structured as a connected, directed, acyclic graph.

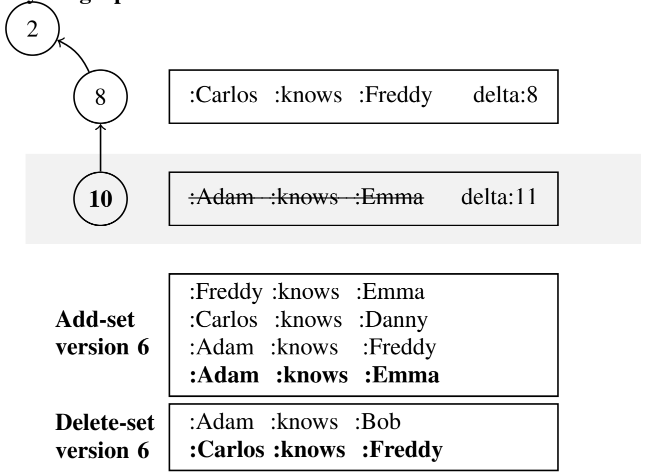

Figure 3: Merging versions 6 and 10 causes triple conflicts, because version 6 contains a deletion which has been added in 8. and an addition which has been deleted in 10.

3.3 Merging strategy

In Git repositories, it is fairly common to merge branches that have diverged and should, naturally, be supported in a triple version control system as well. Merging occurs when two parties start from an initial version to add two different features, and in the end, both features need to be incorporated in the end result. For instance, in Figure 2, the commit chains 2–4–6 and 2–8–10 both start from commit 2, and are merged in commit 12.

In order to merge two branches, we need to determine which triples have been added or deleted in *any* of the branches. This is done by resolving the deltas of each chain, respecting the dependencies (*e.g.*, deletion after addition results in deletion). In practice, deltas are resolved in a similar way as in the context of query execution, described in Section 3.1. Here, we will keep both the delete-set and add-set of both branches, and unite them.

Source code version control needs to maintain the line order of each file. The used *merging strategy* asures this by determining when merging causes conflicts, and how they can be resolved. Since triples have no order, this strategy is up for discussion. For instance, conflicts can be handled on a semantic level (as mentioned in Section 5) or on a lower, more naïve level. For simplicity reasons, we only discuss a naïve merging strategy below.

A *merge conflict* arises when a triple is added on one branch but deleted on another, or vice-versa. This is demonstrated in Figure 3. If such a conflict arises, we must ask the user to resolve each conflicting triple manually, since we cannot make any assumptions on the semantics.

If there are no conflicts (or if all conflicts have been resolved), the merge can be made. A merge will have two parents and will simply inherit all non-conflicting triples. Since all conflicts must be explicitly resolved, they are added to the commit as part of regular add-sets and delete-sets. As a result, when a query is executed, the triple either come from (i) branch A xor branch B if only one branch contains the triple, (ii) branch A and branch B if both branches contain the triple, (iii) from the merge commit's add-set if there was a conflict.

3.4 Supporting blank nodes

Blank nodes pose an issue when it comes to modifying triples and this is no different with versioning. Since blank nodes cannot be identified, deleting them is not trivial. Therefore, we handle blank nodes in the following way. Added blank nodes will be part of the add-set of the commit, as would any other triple. When deleting blank nodes, (i) all triples that match the triple pattern in the delete-set are deleted, or (ii) specific triples with blank nodes are deleted when the internal identifier, as added by the triple store, is used. We refer to future work for a more in-depth study on this issue.

3.5 Versioned querying using virtual graphs

Current triple stores allow data to be stored in different graphs. A graph can be queried separately by specifying its URI in the FROM clause of the SPARQL query. In our approach, we will expose different versions of a graph as virtual subgraphs. Each version refers to a resolved path of deltas. Therefore, the URI is composed of the graph URI and the selected version identifier, as described in the provenance. For example, the versions $V_{2v_n} = \{2v_i, \dots, 2v_j\}$ of graph *http://example.com/graph* will be available as virtual graphs *http://example.com/graph/versions/*$2v_i, \dots, 2v_j$. The SPARQL query in Listing 2, could therefore select the version from Figure 1.

```
SELECT *
FROM <http://example.com/graph/versions/aBcDeFg>
WHERE {?s ?p ?o}
```

Listing 2: A SPARQL query executed on the endpoint *http://example.com/sparql/*, querying version aBcDeFg, corresponding to delta 6, of the graph *http://example.com/graph/*

Although this eases version selection, querying a specific version will not always be desirable. The URIs of different versions can be unknown, or the user wants to query the latest version validated by the triple store manager. References are introduced. The first kind of references are *tags*. A tag gives an alias to a version. The second kind of references are *branches*. A branch refers to the last version of a certain branch in the tree and give an alias to this branch. The *master* branch is automatically created when a new R&Wbase graph is initialized. A special kind of a reference is the *HEAD*. It points to another reference, which will be used as the default branch to read from and write to. References can be used as if they were version numbers. In Listing 2 it is possible to substitute *http://example.com/graph/versions/6* with *http://example.com/graph/versions/master* when 6 would be the last version of the master branch at that time. This abstracts our approach to the SPARQL endpoint, which can be queried the same way as with any other triple store.

```
SELECT *
WHERE {?s ?p ?o}
```

Listing 3: A SPARQL query with the same result as Listing 2 when executed on *http://example.com/graph/versions/6/sparql/*

In Listing 3, instead of using the *FROM* clause to query the right version of the data, another SPARQL endpoint can be used, dedicated to a certain version. When version 6 would be the master branch, and the HEAD would refer to this one, the same result can be retrieved when querying *http://example.com/graph/*.

To conclude this section, we note that each virtual graph has its complete provenance graph used internally. Therefore, it is automatically published (e.g., in a HTTP Link-header) with every query response, allowing the user to obtain trust over the results.

4. PROOF OF CONCEPT

In this section, we discuss the context of implementation as a proof of concept. In the previous sections, we have introduced an novel approach for providing distributed triple version control in triple stores. A main contribution is the reduction of storage space, which we demonstrate as follows. We sequentially add m commits $C_m = \{c_1, \dots, c_m\}$ to an initial graph G with n triples. Each commit c_i adds an amount of triples a and deletes an amount of triples d. For a naïve, graph-per-version approach, the resulting amount of triples t_A is equal to:

$$\begin{aligned} t_A &= n + (n+a-d) + ((n+a-d)+a-d) + \dots \\ &= n + m \times n + \textstyle\sum_{i=1}^{m} i \times (a-d) \\ &\approx n + m \times n + m^2 \times (a-d) \end{aligned}$$

For our approach, the resulting amount of triples n_B would be:

$$t_B = n + m \times (a+d)$$

In almost all cases, a and d will be tiny compared to n, resulting into serious data reduction. Only in case a and d are similar in magnitude to n (or larger), or when d is considerably larger than a, our approach requires more storage space in the long haul. However, this scenario seems unlikely. Furthermore, this can be resolved by archiving several versions after rebasing, which is described later on.

Another contribution is the support for separately accessible versions, which are resolved at query-time and identified by virtual graphs. We build an interpretation layer that interacts with a triple store. It works triple store independent, but requires the support for *quads*. The interpretation layer is mainly responsible for the filter operations resolving the commit paths. A naïve implementation could rewrite each query to include the right filter operations and let the SPARQL processor resolve the right version. Although this would lead to a fairly easy implementation, filter operations in SPARQL increase execution time tremendously. Therefore, we choose to filter the triples present in the requested version beforehand (e.g., as part of a odbc connection), thus enabling the following optimisations:

Caching: Frequently requested versions, commits or branches can be cached, speeding up the query time.

Indexing: The metadata describing the different branches can be indexed, speeding up the filter operations.

Rebasing: Currently, we only discussed deltas being relative to the initial delta, however, this will not always be optimal. In this case, the initial delta is the *base node*. Every delta has a path leading back to the base node. This path can not only progress (when the start commit has a lower identifier), but regress (when the start commit has a higher identifier). We can move the base node further up the chain. This avoids having to go to far back in time to resolve a version. When a rebasing occurs, all paths leading from the new base node to the former base node are reversed. Rebasing can restructure deltas to obtain the lowest necessary amount of triples. Also, when storage becomes an issue, deltas added before the base node can be archived (*e.g.*, serialised and compressed) or deleted, lowering the number of active triples.

A final contribution is the integration of provenance on operation level. Changes are directly described as provenance, and no longer depend on descriptions by agents or persons.

5. FUTURE WORK

Just like Git for software, R&Wbase may create new possibilities in the context of data management. In this section, we discuss future work, divided into two main parts: *interfaces* and *use cases*.

5.1 Interfaces

Users will need to be able to interact with R&Wbase, therefore, a study of possible interfaces is a logical step. Since we are building a Linked Data solution for the Web, choosing SPARQL, as described in Section 3.5, as a first interface is obvious. We however also need a command line interface to manage distributed R&Wbase graphs and a HTTP interface to create a Linked Data Platform.

Just like Git, a set of commands is needed by which the data can be managed in a distributed way. The base command could be *rwb*, an acronym for R&Wbase, with a set of subcommands inspired by the Git workflow: *commit, pull, push, clone, merge, diff, rebase, tag, branch, init*... For instance, cloning a dataset will result in ingesting data in your own local triple store, pulling and pushing will be used to synchronize changes with other copies. Merging, as discussed in 3.3, will combine the changes in different branches. For now, we have handled this issue rather naïvely. In future research, we will work towards an intelligent way to resolve merge conflicts. An example could be semantic merging, where ontology information is used to create a merge that makes the most sense. Next, rebasing, as discussed in 4, can select a more optimal delta as reference point, to keep the other deltas as small as possible and *rwb diff* will show all uncommitted changes to the graphs in a human readable way.

Next to SPARQL SELECT operations, SPARQL 1.1 also describes an interface to update triples [5]. With the *INSERT*, *DELETE*, *DELETE/INSERT*, *LOAD* and *CLEAR* operations, triples can be created, updated or deleted. In future work, we describe an interpretation layer translating SPARQL queries to the underlying quad-store. Furthermore, An HTTP interface is required as well, making the chosen URIs resolvable. At the time of writing, there is a working draft for a Linked Data Platform (LDP) [10]. Future work will describe how LDP can manage the different versions of the data.

5.2 Use cases

R&Wbase can be the driver behind a lot of new use cases. Keeping these use cases in mind, R&Wbase will have to implement other features as well, which future research will have to discuss. For instance, based on *trust* and provenance, an application can decide which version of the data is most optimal for a certain user. A second example is storage of data valid in the future. For example, if we want to store a lawbook in R&Wbase: references (tags) could be used to query a certain version of the law book at a certain time. But also, accepted laws will only be introduced after a year and thus, querying in the future needs to be implemented. A third example is of course the Open Data Ecosystem where a workflow needs to be defined for *Data Governance*. Branches can be merged manually (comparing every triple), but also automatic merging, based on certain parameters, or resolving conflicts are possible.

6. CONCLUSION

In this paper, we described how triple stores can support upcoming Read/Write infrastructures on the Semantic Web, using distributed triple version control. Instead of copying entire graphs to provide versioning, triples are stored as consecutive deltas. Each delta only stores an add-set and a delete-set relative to the previous version. We have shown that our approach shows a significant reduction in storage space. The number of stored triples is now relative to the delta size, instead of the graph size, which is much smaller in most cases. Furthermore, we explained how an interpretation layer, interacting with a *quad*-store, can provide access to different versions. We describe how the different deltas are stored in the same graph, and efficiently distinguished using their context value. Added triples of a delta $2y$, are stored with the context value $2y$, while deleted triples are stored with context value $2y+1$. We have shown how different versions can be easily resolved by extracting triples whose highest occurring context value is even. The use of caching, indexing and rebasing (changing the base node of each delta path) was discussed for further optimisation of version resolving. Then, we demonstrated how deltas are made by adding *commits*, which consist of a delta and associated metadata. Each commit holds a reference to its parent and is therefore used to resolve a specific version of the graph. We have shown how commits allow versions to exist in parallel, creating different *branches*. The merging of different branches was discussed, and how possible conflicts can be resolved. Furthermore, we argued how commits are equivalent to the notion of provenance, which can be directly described in PROV-O. This includes provenance at the operation level, avoiding human or agent interference, which can be automatically be published to provide trust. Finally, we described how each version can be accessed separately at query-time, using virtual graphs in SPARQL.

7. ACKNOWLEDGEMENTS

The research activities described in this paper were funded by Ghent University, iMinds, the Flemish department of Economics, Science and Innovation (EWI), Tourism and Recreation West-Flanders (Westtoer), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research-Flanders (FWO-Flanders), and the European Union.

8. REFERENCES

[1] The Open Archives Initiative Protocol for Metadata Harvesting, 2004.

[2] S. Cassidy and J. Ballantine. Version control for RDF triple stores. In *ICSOFT (ISDM/EHST/DC)'07*, pages 5–12, 2007.

[3] S. Coppens, R. Verborgh, M. Vander Sande, D. Van Deursen, E. Mannens, and R. Van de Walle. A truly Read-Write Web for machines as the next-generation Web? In *Proceedings of the SW2022 workshop: What will the Semantic Web look like 10 years from now?*, Nov. 2012.

[4] R. Cyganiak, A. Harth, and A. Hogan. N-Quads: Extending N-Triples with Context, Novemer 2009.

[5] P. Gearon and S. Schenk. SPARQL 1.1 update. W3C proposed recommendation, W3C, Nov. 2012. *http://www.w3.org/TR/2012/PR-sparql11-update-20121108/*.

[6] Y. Gil, J. Cheney, P. Groth, O. Hartig, S. Miles, L. Moreau, and P. P. da Silva. Provenance XG Final Report, Dec. 2010.

[7] 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, 2012.

[8] R. Polock. Building the open data ecosystem, 2011.

[9] B. Schandl. Replication and versioning of partial RDF graphs. In *Proceedings of the 7th international conference on The Semantic Web: research and Applications - Volume Part I*, ESWC'10, pages 31–45, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] S. Speicher and J. Arwe. Linked Data Platform 1.0. W3C working draft, W3C, 2013. *http://www.w3.org/TR/ldp/*.

[11] L. Torvalds. git(1) manual page, 2013.

[12] S. van Hooland, R. Verborgh, M. De Wilde, J. Hercher, E. Mannens, and R. Van de Walle. Evaluating the success of vocabulary reconciliation for cultural heritage collections. *Journal of the American Society for Information Science and Technology (JASIST)*, 64(3):464–479, Mar. 2013.

[13] M. Völkel and T. Groza. Semversion: Rdf-based ontology versioning system. In *Proceedings of the IADIS International Conference WWW/Internet 2006 (ICWI)*, page 44, 2006.