

Scalable Semantic Version Control for Linked Data Management

Claudius Hauptmann, Michele Brocco, and Wolfgang Wörndl

Technische Universität München, 85748 Garching, Germany
{hauptmac,brocco,woerndl}@cs.tum.edu

Abstract. Linked Data is the Semantic Web’s established standard for publishing and interlinking data. When authors collaborate on a data set, distribution and tracking of changes are crucial aspects. Several approaches for version control were presented in this area each focusing on different aspects and bounded to different limitations. In this paper we present an approach for semantic version control for dynamic Linked Data based on a delta-based storage strategy and on-demand reconstruction of historic versions. The approach is intended for large data sets and supports targeted and cross-version queries. The approach was implemented prototypically on top of a scalable triple store and tested with a generated data set based on parts of DBpedia with several million triples and thousands of versions.

Keywords: Linked Data, Version Control, SPARQL, Revision, Query, Provenance, Named Graphs

1 Introduction

Linked Data provides essential mechanisms to efficiently interlink and integrate data using the Resource Description Framework (RDF) as base model [5, 9]. RDF stores information as directed graph. Edges are defined by triples consisting of subject, predicate and object and nodes are defined implicitly through the edges and are referenced by URIs. Edges can be grouped to named graphs to facilitate the administration or to store additional information by assigning a context, which transforms triples to quads. SPARQL (SPARQL Protocol And RDF Query Language) can be used as query language for Linked Data using pattern matching, filtering, aggregation and even distributed query execution to query several data sources at once.

A missing feature not covered by the Linked Data standard so far is version control. Especially when several authors are involved (which is obviously the case for the data amounts addressed by Linked Data) tracking and distribution of changes and rolling back to previous revisions are crucial aspects for any kind of data management [5, 9, 6]. Recent research projects created several approaches for version control of Linked Data with focus on different aspects. They cover versioning of data of OWL ontologies, lightweight RDFS ontologies and Linked Data, support different workflows and enable knowledge workers to run different

query types on versioned data. Some are limited in scalability regarding the number of triples, the number of versions or because of space efficiency. Some solutions hide the version information from the Linked Data layer preventing the access of version information by SPARQL queries.

Since Linked Data is designed to handle and publish large data sets we focus on scalable semantic version control. This comes with new limitations, especially the variety of query types that can be handled. Because of the amounts of data we want to handle, we use a delta-based strategy. Triples of historic versions that are used for query evaluation are reconstructed on-demand. This comes at the cost, that global queries (which require the whole dataset for a specific version to be constructed) can hardly be supported. The construction of versions occurs very frequently, thus the version construction performance is the critical factor[6].

The goal of this paper is to propose an approach for scalable semantic version control for dynamic Linked Data based on partial on-demand reconstruction of historic versions. We focus on query optimization for targeted queries on random versions stored in a delta-based, distributed storage. Like Graube et. al. [5] we want the version control information itself to be accessible by SPARQL queries.

We prototypically implemented a semantic version control system on top of a scalable triple store and analyzed query execution plans and their performance using several million triples. We optimized queries accessing historical information and added a module to the query engine that constructs and caches relevant triples on-demand. The implemented approach was evaluated in terms of space efficiency and query performance.

Section 2 of this paper shows related work, Section 3 describes our approach and Section 4 a performance test. Section 5 closes with a summary and an outlook to future work.

2 Related Work

Stefanidis et. al. [10] discuss storage strategies and recommend hybrid strategies over pure version-based or delta-based strategies. They distinguish between modern, historical and cross-version queries, global vs. targeted queries and version-centered vs. delta-centered queries. Graube et. al. [5] show an approach for semantic version control targeting medium sized data sets supporting both version-specific queries and cross-version queries. They use a delta-based and a version-based store holding the latest version. By applying changesets, versions are temporarily adapted on the fly to the version specified by the query. The approach is limited by the number of changes that can be applied to an index on the fly. Tzitzikas et al. [12, 8] develop storage index structures based on partial orders that store several overlapping versions of RDF datasets. Im, Lee and Kim [6] argue that this approach is not scalable if a query needs to fully construct a specific version. They use a relational database to store deltas separately in a delete and an insert table. They construct logical versions on the fly using a SQL statement that joins the original version and the relevant delta tables. To

reduce overhead they introduce aggregated deltas. Im et. al [7] use a hypergraph that exploits hyperedges and vertices for RDF version management. In contrast to Tzitzikas et al. [12, 8] versions do not have to be constructed per query since the hypergraph allows to store the relations between edges and versions. The approach is limited in scalability since a combination of several million edges and several thousand versions is demanding in terms of space and the addition of a version will also be time consuming. Auer and Herre [1] focus on ontology evolution by tracking and classifying changes made to RDF stores. Cassidy and Ballantine [4] use a delta-based storage and a working storage backed by relational databases on both server and client that synchronize changes. Sande et al. [9] store deltas in a quad-store. Versions can be queried by SPARQL through virtual graphs. An interpretation layer transforms SPARQL queries to reconstruct the versions in the quad store on the fly to a triplestore. SemVersion [13] is a RDF-based ontology versioning system supporting historic queries. The information about versions is hidden from SPARQL queries and the space overhead limits scalability. Lee and Conolly [2] focus on comparing and updating RDF graphs by generating sets of differences and propose an update ontology for patches of RDF files. We are looking for a solution for large data sets that recreates historic versions on-demand and therefore accept limitations regarding global queries [10] which require the whole data set for a version to be constructed. Our focus is different in contrast to previous work and needs its own optimizations which are discussed and tested in the following sections.

3 On-demand Version Reconstruction

Like Cassidy and Ballantine [4] and Graube et. al. [5] we handle versioning on the level of complete graphs and we model the delta-sets as Linked Data. Each version of a graph is referenced by a *commit* and, except the first one, each commit *references (ref)* a *previous commit (prev)*. Since several commits can reference the same previous commit it is possible to work on *branches* in parallel. Merging two branches is done by creating a commit that is referencing two previous commits. All edges are defined by triples and as these triples belong to the same graph, we can store them as quads and use the context information as *identifier* for the triple. The commits either add or remove triples stored as edges that reference these triples having a predicate of type *add* or *delete*. Merge commits store only those triple modifications that are ambiguous and would lead to conflicts. We store *branches* and *tags* as edges that link a branch URI to their current commit. The branches and tags are referenced by an edge connecting them to a *graph*. Figure 1 shows an example instance of this model.

To access historic versions we use virtual graphs that specify the version we want to use in our query. In the following targeted query [10] we use data in a branch called *branch1* to load triples of the graph *http://graph1* by accessing the virtual graph *http://graph1/branches/branch1*:

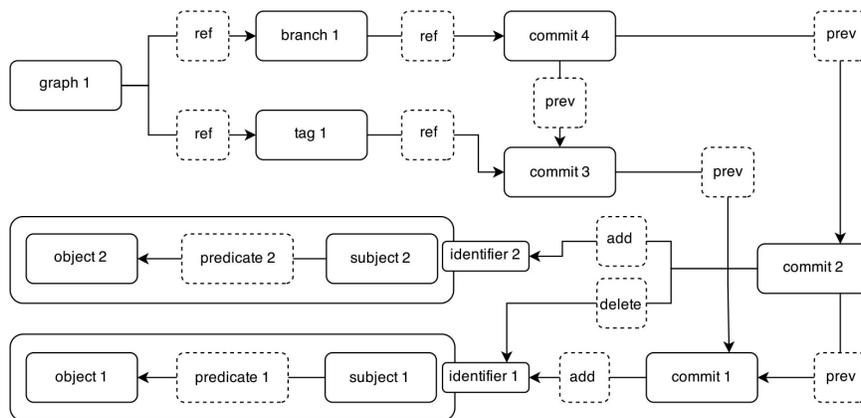


Fig. 1. delta-based storage schema

```

SELECT ?object WHERE {
  GRAPH <http://graph1/branches/branch1> {
    ?subject :predicate2 ?object .
  }
}

```

As delta-based storages store only differences between versions, we cannot run this query directly. One way solving this issue would be to reconstruct the data and to run the query afterwards. Another way is to adapt the query to the actual data structure by rewriting the original query in order to use the delta-based storage:

```

SELECT ?object WHERE {
  GRAPH ?identifier {
    ?subject :predicate2 ?object .
  }
  GRAPH <http://graph1> {
    <http://graph1/branches/branch1> versioning:references ?branchCommit .
    ?branchCommit versioning:hasPrevious* ?addCommit .
    ?addCommit versioning:adds ?identifier .
    FILTER NOT EXISTS {
      ?branchCommit versioning:hasPrevious* ?deleteCommit .
      ?deleteCommit versioning:hasPrevious* ?addCommit .
      ?deleteCommit versioning:deletes ?identifier .
    }
  }
}

```

Like predicted by Sande et. al. [9], we measured that this query is very time consuming because of an computational overhead for this type of queries. First, most triple stores reconstruct the whole desired version in the first step and then run the original query that in fact reuses only small parts of this very large intermediate result. We solved this by giving query hints to the query engine. Second, the arbitrary path length operators that traverse the commit graph (like

*versioning:hasPrevious**) are executed very often - once per triple per possible solution, to check whether the binding set is part of the version. Third, the associations between triples and versions are not cached. This is especially useful when joins are used in queries and a triple is matched against other triples several times.

Once per query we load the commit graph into an in-memory index which takes few seconds for 28,000 commits. Instead of using arbitrary path length operators we create and use our indexes to check whether the triples used in possible results are in fact part of the specified version and the result should be returned. Query engines of scalable triple stores work pipelined [14]. While the first iterator is performing an index scan it sends first results to the following iterator (e.g. a hash join). We work on chunks of up to 100 triple identifiers for which we load the associations to commits into an index. We traverse the commit graph once per chunk and select and return the triple identifiers that are part of the specified version.

The proposed approach was prototypically implemented and is based on Blazegraph (formerly Bigdata) [11], a scalable, distributed triple store which implements the Sesame API [3]. Currently we did not automate the rewriting process. To access historical data we implemented a custom service that is called from queries via the *SERVICE* keyword as a virtual service. Since Blazegraph contains several virtual services that extend the functionality of SPARQL, this seemed to be an appropriate way. The service gets the version identifier from the original query as triple patterns inside the service query and the triple identifiers via bindings (Bindings are comparable to query parameters in SQL). For each triple pattern we check the version association and use a hash table to cache this information. If the triple is part of the version we return that triple identifier as a binding to the original query that called the virtual service. The rewritten query from the last section with our approach changes into:

```
SELECT ?object WHERE {
  GRAPH ?identifier {
    ?subject :predicate2 ?object .
  }
  SERVICE versioning:service {
    versioning:version versioning:value <http://graph1/branches/branch1> .
    versioning:binding versioning:value ?identifier .
  }
}
```

4 Performance Tests

Widely used metrics for evaluation of version control concepts are response time and storage space consumption [5]. We measured the response time for complete queries as well as the response time for single steps of our approach. We ran each query 100 times and calculated average durations. First, we measured the duration for the generation of the commit graph index which is built once per query. Second, we measured the duration for the generation of an index storing the relationships between commits and triple identifiers. Third, the duration and

the average path length (average number of edges traversed per triple identifier) for the graph traversal that checks which triples are part of the specified version. The triple store we used called the virtual service we created once per chunk of 100 triple identifiers. We measured the storage space consumption by cumulating the file sizes of the index segments rather than calculating the number of triples. As dataset we used the English mapping-based types of DBpedia which contains 28,031,852 triples (release 2014). We generated 4 delta-based datasets with 100, 1000, 10,000 and 100,000 triples per commit. The changes were distributed equally to the commits and the history consisted of a single timeline with one branch. We also ran the test queries on a triple store which contained the latest version without any version control as baseline. As test queries we loaded the list of types assigned to Slovenia and the list of instances of type country from the latest commit which contains all triples:

```
SELECT * WHERE { <http://dbpedia.org/resource/Slovenia> ?p ?o }

SELECT * WHERE { ?s ?p <http://schema.org/Country> }
```

Table 1 shows the disk space consumption, table 2 and 3 show the measured durations for the test queries.

#triples/commit	baseline	100	1,000	10,000	100,000
#commits	-	280,319	28,032	2,804	281
Disk space consumption [MB]	2,774	11,237	10,910	10,447	10,438

Table 1. Number of commits and disk space consumption

#triples/commit	baseline	100	1,000	10,000	100,000
Query response time [ms]	48	5,076	2,685	2,434	2,382
Index creation for a graph of commits [ms]	0	4,610	2,355	2,090	2,067
Index creation for a chunk of 100 triples [ms]	0	161	167	173	152
Graph traversal for a chunk of 100 triples [ms]	0	144	13	1	1
average path length	0	279,782	27,980	2,800	282

Table 2. Average durations [ms] for the first test query returning 7 results

The results show that we can run targeted queries on a delta-based storage with on-demand creation of historic versions. The first test query uses an index of the version-based storage of the baseline and is hundred times faster than a delta-based storage. In the second test query a version-based storage would be four times faster than a delta-based storage. The disk space consumption is about four times higher than without version control. Our approach is limited by the number of versions that increase the time spent for graph traversal and by the size of the results or intermediate results. Global queries (like looking for

#triples/commit	baseline	100	1,000	10,000	100,000
Query response time [ms]	2,105	14,094	7,508	6,878	6,689
Index creation for a graph of commits [ms]	0	4,597	2,327	2,071	2,051
Index creation for a chunk of 100 triples [ms]	0	3,752	3,564	3,878	3,620
Graph traversal for a chunk of 100 triples [ms]	0	4,386	327	44	6
average path length	0	276,515	27,629	2,768	278

Table 3. Average durations [ms] for the second test query returning 3,108 results

the average number of friends [10]) will be faster on a version-based approach. If these query types are important, a version-based approach is preferable over a delta-based approach. If the number of versions used for global queries is limited, a hybrid approach could be used or specific versions could be extracted into additional triple stores. We did not analyze the behaviour of the approach under a more realistic history containing several branches, merges and several changes on the same value (e.g. same subject and predicate) as well as more complex queries with several graph patterns.

5 Conclusion and Outlook

In this paper we present an approach for scalable semantic version control for Linked Data based on a delta-based storage and on-demand reconstruction of historic versions. Versions are handled on a graph level and random versions of graphs can be accessed transparently within SPARQL queries through virtual graphs. We showed that targeted queries on random versions that reconstruct necessary parts of the datasets on-demand are possible. The optimization we propose is based on the query execution order, using in-memory indexes and caching of intermediate results, that are frequently used within a query.

There are several aspects we did not analyze yet. The index for a graph of commits can be cached and reused for several queries to the same graph which would save a remarkable amount of time. In our implementation we did not operate on the internal identifiers or the storage engine but on the URIs which have to be loaded from an additional index which leaves also room for optimization. BlazeGraph also has an optimized storage option for statement identifiers that might save space for the delta-based storage. The size of the indexes that have to be traversed could also be reduced by a hybrid storage strategy. To automate the creation of the materialized versions for a hybrid approach a cost model is necessary to decide which versions to create. This idea is comparable to lazy materialization of indexes for relational databases [15]. We did not implement or evaluate the creation of merge commits which involves checking for conflicts. Yet, we focused on querying random versions of graphs in delta-based storages but did not propose an approach to use SPARQL update queries without a materialized version that has change listeners attached. This is also an important feature that needs further research.

References

1. Auer, S., Herre, H.: A Versioning and Evolution Framework for RDF Knowledge Bases. In: Virbitskaite, I., Voronkov, A. (eds.) 6th Intl. Andrei Ershov Memorial Conference, PSI 2006. pp. 55–69. Springer Berlin Heidelberg, Novosibirsk (2006)
2. Berners-Lee, T., Connolly, D.: Delta: an ontology for the distribution of differences between RDF graphs (2001), <http://www.w3.org/DesignIssues/Diff>
3. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: Proceedings of the First International Semantic Web Conference. Springer Berlin Heidelberg, Sardinia (2002)
4. Cassidy, S., Ballantine, J.: Version Control for RDF Triple Stores. In: Filipe, J., Shishkov, B., Helfert, M., Maciaszek, L. (eds.) Proceedings of the 2nd International Conference on Software and Data Technologies. pp. 5–12. Springer-Verlag Berlin Heidelberg, Barcelona (2007)
5. Graube, M., Hensel, S., Urbas, L.: R43ples: Revisions for Triples - An Approach for Version Control in the Semantic Web. In: Knuth, M., Kontokostas, D., Sack, H. (eds.) 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems. CEUR Workshop Proceedings, Leipzig (2014)
6. Im, D.H., Lee, S.W., Kim, H.J.: A Version Management Framework for RDF Triple Stores. *International Journal of Software Engineering and Knowledge Engineering* 22(01), 85–106 (Feb 2012)
7. Im, D.H., Zong, N., Kim, E.H., Yun, S., Kim, H.G.: A hypergraph-based storage policy for RDF version management system. 6th International Conference on Ubiquitous Information Management and Communication - ICUIMC '12 p. 1 (2012)
8. Psaraki, M., Tzitzikas, Y.: CPOI : A Compact Method to Archive Versioned RDF Triple-Sets (Ic), 1–25 (2010)
9. Sande, M.V., Colpaert, P., Verborgh, R., Coppens, S., Mannens, E., Walle, R.V.D.: R & Wbase : Git for triples. In: Bizer, C., Heath, T., Berners-Lee, T., Hausenblas, M., Auer, S. (eds.) Proceedings of the WWW2013 Workshop on Linked Data on the Web. pp. 1–5. CEUR Workshop Proceedings, Rio de Janeiro (2013)
10. Stefanidis, K., Chrysakis, I., Flouris, G.: On Designing Archiving Policies for Evolving RDF Datasets on the Web pp. 43–56 (2014)
11. Thompson, B., Personick, M., Cutcher, M.: The Bigdata RDF Graph Database. In: Wagner, A., Hose, K., Schenkel, R. (eds.) *Linked Data Management*, chap. 8, pp. 193–237. CRC Press, Boca Raton (2014)
12. Tzitzikas, Y., Theoharis, Y., Andreou, D.: On storage policies for Semantic web repositories that support version. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) 5th European Semantic Web Conference. pp. 705–719. Springer Berlin Heidelberg, Tenerife (2008)
13. Völkel, M., Groza, T.: SemVersion: An RDF-based ontology versioning system. In: Proc. 5th IADIS International Conference on WWW/Internet. IADIS Press (2006)
14. Wang, X., Tiropanis, T., Davis, H.C.: LHD: Optimising Linked Data Query Processing Using Parallelisation. In: Bizer, C., Heath, T., Berners-Lee, T., Hausenblas, M., Auer, S. (eds.) Proceedings of the WWW2013 Workshop on Linked Data on the Web. vol. 996. CEUR Workshop Proceedings, Rio de Janeiro (2013)
15. Zhou, J., Larson, P.A., Elmongui, H.G.: Lazy Maintenance of Materialized Views. In: Proceedings of the 33rd international conference on Very large data bases. pp. 231–242. VLDB Endowment, Vienna (2007)