



Master's Thesis

Versioning Linked Datasets

Towards Preserving History on the Semantic Web

Author

Paul Meinhardt

744393

July 13, 2015

Supervisors

Magnus Knuth and Dr. Harald Sack

Semantic Technologies Research Group

Internet Technologies and Systems Chair

Abstract

Linked data provides methods for publishing and connecting structured data on the web using standard protocols and formats, namely HTTP, URIs, and RDF. Much like on the web of documents, linked data resources continuously evolve over time, but for the most part only their most recent state is accessible. In order to investigate the evolution of linked datasets and understand the impact of changes on the web of data it is necessary to make prior versions of such resources available. The lack of a common “self-service” versioning platform in the linked data community makes it more difficult for dataset maintainers to preserve past states of their data themselves. By implementing such a platform which also provides a consistent interface to historic information, dataset maintainers can more easily start versioning their datasets while application developers and researchers instantly have the possibility of working with the additional temporal data without laboriously collecting it on their own.

This thesis, researches the possibility of creating a linked data versioning platform. It describes a basic model view for linked datasets, their evolution and presents a service approach to preserving the history of arbitrary linked datasets over time.

Zusammenfassung

Linked Data beschreibt Methoden für die Veröffentlichung und Verknüpfung strukturierter Daten im Web mithilfe standardisierter Protokolle und Formate, nämlich HTTP, URIs und RDF. Ähnlich wie andere Dokumente im World Wide Web, verändern sich auch Linked-Data-Ressourcen stetig mit der Zeit. Zumeist ist jedoch nur ihr aktueller Zustand zugänglich. Um die Evolution von Linked Datasets untersuchen zu können und wie sich Änderungen auswirken, ist es notwendig, frühere Versionen solcher Ressourcen verfügbar zu machen. Das Fehlen einer frei nutzbaren Versionierungsplattform in der Linked-Data-Gemeinschaft erschwert es den Veröfentlichern von Datensätzen geänderte Daten zu archivieren. Durch die Implementierung einer solchen Plattform, welche eine konsistente Schnittstelle zu den historischen Informationen bietet, können die Bereitsteller von Daten leichter mit der Versionierung beginnen. Anwendungsentwickler und Wissenschaftler erhalten die Chance mit den zusätzlichen zeitlichen Daten zu arbeiten, ohne sie selbst umständlich sammeln zu müssen.

Diese Arbeit untersucht die Möglichkeit der Schaffung einer solchen Linked Data Versionierungsplattform. Sie beschreibt eine grundlegende Modellansicht für Linked Datasets, deren Evolution und präsentiert einen serviceorientierten Ansatz zur Erhaltung der Historie beliebiger Linked Datasets.

Acknowledgements

I am grateful to my supervisors Dr. Harald Sack and Magnus Knuth for their guidance and support.

I thank Magnus Knuth especially for fruitful discussions and his comments on drafts of this thesis.

Contents

1	Introduction	1
1.1	Contributions	4
1.2	Thesis structure	4
2	Background	7
2.1	Linked Data	7
2.2	RDF	8
3	Motivation	13
3.1	Linked Data Publishing Workflows	13
3.2	The Need for Linked Data Versioning	15
3.3	Versioning Practice	16
4	Related Work	19
4.1	RDF Change Descriptions	19
4.1.1	Change Vocabularies and Ontologies	19
4.1.2	Patch Formats	20
4.2	HTTP-based Version Access	21
4.3	RDF Store Versioning	22
4.3.1	Version Control for RDF Triple Stores	22
4.3.2	R&Wbase	22
4.3.3	R43ples	23
4.3.4	Apache Marmotta	24
4.4	Versioning Services	24
5	Approach	27
5.1	A Model for Linked Datasets	29
5.2	Possible Changes in Linked Datasets	30

Contents

5.3	Querying Linked Dataset History	30
6	Investigated Alternatives	33
6.1	Triplestore Extension	33
6.1.1	Native Triplestores	33
6.1.2	Virtuoso Universal Server	34
6.2	VCS-based Implementation	35
6.2.1	Git	35
6.2.2	Mercurial	38
6.3	Summary	40
7	Implementation	43
7.1	System Overview	43
7.2	Memento API	43
7.3	Storage Model	47
7.4	Push API	52
7.5	User Interface	53
7.6	Integration Points	54
8	Evaluation	57
8.1	Test setup	57
8.1.1	Sample Dataset	57
8.1.2	Test Infrastructure	58
8.2	Push Performance	59
8.3	Storage Analysis	61
8.4	Revision Retrieval	62
8.5	Remarks	64
9	Future Work	67
9.1	Platform Enhancements	67
9.2	Evaluation of Alternative Backends	67
9.3	Monitoring Service	68
9.4	Analysis Tools	69
10	Summary	71

Contents

Literature

73

Preliminaries

This work includes examples and figures that contain URIs from common namespaces. For readability and clarity we will use the following namespace prefixes to abbreviate long URIs (prefix on the left, replacement right):

```
xsd:      http://www.w3.org/2001/XMLSchema#  
rdf:      http://www.w3.org/1999/02/22-rdf-syntax-ns  
dbp:      http://dbpedia.org/resource/  
dbo:      http://dbpedia.org/ontology/  
dbpp:     http://dbpedia.org/property/  
umbel:    http://umbel.org/umbel/rc/
```


1 Introduction

The key observation that lies at the very core of the linked data effort is that data is useful beyond small groups of collaborating people in research and other communities. Usually however, data is not made available in a way that allows it to be referenced or queried in a consistent manner. That is mostly because data is published in a variety of application- or domain-specific formats and protocols which prevent others from using it. While the world wide web, the “web of hypertext”, has become an incredibly important source of information today, it lacks methods required for querying and processing information and deriving clues from it in an automated manner with proper support from software agents. Therefore, linked data proposes a generic way to express information and principles for structured data publishing allowing data to be interlinked and integrated into a global knowledge graph, a “web of data”.

Much like the web of documents, this global knowledge graph is in a steady state of flux, constantly changing. With the status quo in linked data publishing, data is largely replaced with updated versions. Generally only the most recent working state is kept and manipulated directly, discarding any overwritten or removed data as updates are performed.

This practice results in a substantial loss of information that is routinely kept in many modern web applications. Wikipedia, for instance, stores the revision history for all of its articles.

Being able to access such histories is useful for a number of reasons:

Version References and Data Consistency In a distributed graph like the web, where different authorities are responsible for distinct parts, referenced data may change or become unavailable at any time without the knowledge of those

referencing it. As common practice with software dependencies, authors may instead wish to explicitly link to a specific revision of a resource, an ontology or vocabulary for instance.

Change Inspection Data users may further wish to investigate deltas between revisions, determining what kind of changes were made. For an overview, the amount of added, removed or unchanged statements hint at how significant a change was with regard to the overall amount of data. More detailed inspection might reveal updated facts or changed semantics.

Data Quality Assessment The possibility of looking at the history of an information resource also opens up opportunities for assessing the quality of the data. In particular, temporal attributes like recency, volatility, and timeliness [1] may be evaluated based on revision information. These help data consumers in understanding whether or not a dataset is actively maintained and whether they should use it.

Dynamic Processes While time-series observations can be represented explicitly, many data sources only provide single, up-to-date values. Wikipedia pages for companies, for instance, only provide the current numbers for revenue; country pages only give the most recent population numbers. For the same reason, DBpedia contains just these values. In order to study dynamic real-world processes however, being able to look at the development of such indicators is crucial.

Data Dynamics Finally, understanding the evolution of the web of data itself requires recording state changes. Only this way, we can later identify patterns in changes and understand how they propagate, see which new resources are referenced and how they grow.

Without preserving the histories of linked datasets many of such insights are impossible. Yet there is no singular established practice for versioning in the linked data community. Most datasets only provide occasional, versioned releases in the form of dumps, others do not have a versioning strategy at all.

From the perspective of a linked data publisher, a lot of effort goes into maintaining the dataset itself and also the needed infrastructure. Even without the

additional task of recording a revision history, a large fraction of the linked data web suffers from poor availability and other technical problems [2]. When data publishers go the extra mile for archiving past dataset states, different solutions expose different, non-standard interfaces for version access, limiting their usefulness.

For linked data consumers on the other hand, the lack of a central repository for past dataset states frequently means they have to gather such information on their own. When they do, their results are not always made public. Some will provide dumps, but those are generally not very discoverable and do not allow granular access. You need to know how these files are organized and then download the appropriate dump as a whole. From a data consumer perspective, a consistent access mechanism is missing.

This thesis proposes an approach for preserving the entire state of collections of linked data resources as the basis for retrospective investigations of the linked data graph. In particular, this thesis introduces an approach for separating the concerns of linked data publishing and the recording of dataset history. It specifically aims at relieving dataset publishers from the task of maintaining their own versioning solution and infrastructure. At the same time, the proposed approach provides improved discoverability of prior resource states and version access using standard protocol mechanisms.

The open nature of the approach allows anyone to create and share recordings of datasets. Ideally, dataset maintainers will submit changes to an official repository for their own dataset. For datasets where this is not the case, others can push change information from existing linked data crawling and monitoring tools.

The approach enables fine-grained access to separate dataset resource states through time-oriented queries. To that end, it uses a consistent time-travel interface for all versioned datasets that aligns well with linked data principles.

An implementation of our approach is provided as an open-source project written in Python, with a running instance of the service readily available for try-out and suggestions.

Through the creation of an archive for historic linked dataset information, we hope to gradually cover a significant portion of the web of data and make past dataset states available for new applications and further research on the dynamics of datasets.

1.1 Contributions

The primary goal of this work is to research the possibility of creating a linked data versioning platform. The main contributions of this thesis are:

- An approach to versioning datasets based on linked data principles (chapter 5) and an investigation of possible realization paths (chapter 6).
- A design for a linked data archival platform providing interfaces for submitting snapshots of linked data resources as well as a time-based access protocol for resource revisions, backed by a storage model which balances the requirements for storage space and retrieval time (chapter 7).
- An implementation of the presented design as a Python web application which can be used to provide linked data archival as a service to dataset publishers.

1.2 Thesis structure

The remainder of this thesis is structured as follows:

Chapter 2 briefly introduces the fundamental concepts and terminology used throughout this thesis.

Chapter 3 illustrates typical linked data publishing workflows, use-cases for linked data versioning and issues with common versioning practice.

Chapter 4 discusses related work in the area of versioning semantic data.

Chapter 5 then develops the requirements for a linked data archival platform.

Chapter 6 documents alternative implementation approaches towards a ver-

sioning service, investigates their characteristics and limitations.

A service design and working implementation is described in chapter 7 followed by an evaluation of the created system in chapter 8.

Chapter 9 then outlines future perspectives for the developed approach before we conclude with an overall summary of this thesis in chapter 10.

2 Background

This thesis develops an approach to versioning semantic data on the basis of linked data principles. This chapter will briefly introduce the concepts and terminology which form the background of our work.

2.1 Linked Data

The *semantic web* [3] is envisioned as an addition to the world wide web which allows software agents to autonomously discover and interpret available information. As such, the semantic web uses the very same internet infrastructure but relies on explicit, structured knowledge representations to avoid human language ambiguity and other sources of uncertainty present in common web content.

Linked data establishes a set of guidelines for publishing and interlinking information on the semantic web using technology standards. The goal is to enable distributed information resources to be connected and to allow uniform access to make the best use of knowledge worldwide.

The core principles of linked data are [4]:

1. *Use URIs to name things.* In linked data descriptions, abstract concepts as well as real-world objects and living beings receive a unique *URI* so they can be identified and referred to unambiguously.
2. *Use HTTP URIs so they can be looked up.* The use of *HTTP* as a transport protocol allows web clients to retrieve a metadata description associated with a named entity by dereferencing its *URI*.

3. *When someone looks up a URI, provide useful information via RDF (and SPARQL).*
The *Resource Description Framework* (RDF)¹ defines an abstract modelling framework for information representations. The *SPARQL Protocol and RDF Query Language* (SPARQL)² defines another mechanism for efficient remote queries on large RDF datasets. It is considered an extra feature in addition to the basic linked data.
4. *Include links to other URIs, so that they can discover more things.* An important part of modelling knowledge about the world is modelling relationships. URI references help link otherwise separated resource descriptions and provide context information.

Like the name implies, linked data focuses on integrating semantic web resources into a so-called “web of data”. One of the major strengths of this web-based approach: By using retrievable URI references, linked data allows agents to discover and navigate related information resources.

The inherent distributed nature of this approach however means that changes to a resource may have unintended side-effects on those referencing it. Nonetheless, the temporal aspect of linked data is largely neglected in practice. Preservation of past states is not widely handled in a way which allows discovery and navigation of prior resource states.

2.2 RDF

The *Resource Description Framework* (RDF) defines a metadata data model intended for representing information about web resources. In the case of linked data, where web resources represent abstract concepts, real-world objects and people, RDF can be used to describe such entities as well.

RDF represents information about the world in the form of statements. An *RDF statement* is comprised of three components: a *subject*, a *predicate* and an *object*. For that reason statements are also represented and referred to as *triples*.

¹<http://www.w3.org/TR/rdf11-concepts/> (June 19, 2015)

²<http://www.w3.org/TR/rdf-sparql-query/> (June 19, 2015)

A set of triples is called an *RDF graph*. It is common to interpret such a triple-set as a labelled directed graph. The subject and object components of each triple are used to identify graph nodes. The edges of a graph, connecting subject- to object-nodes, are labelled by the corresponding predicate (see figure 2.1).

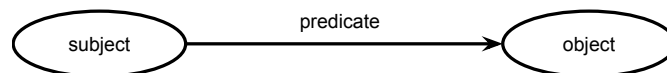


Figure 2.1: Visual representation of an RDF statement with subject, predicate and object as a node and directed-arc diagram.

There may be three distinct kinds of nodes in an RDF graph:

- *IRIs (Internationalized Resource Identifiers)*³,
- *blank nodes* and
- *literals*.

They are collectively known as *RDF terms*.

An IRI uniquely identifies a particular resource by its name. A literal encodes a specific, atomic value, e.g. a string, number or date. A blank node, unlike IRIs or literals, only expresses that something with the given properties exists, without explicitly naming it.

In an RDF statement,

- the subject is either an IRI or a blank node,
- the predicate is an IRI and
- the object is either an IRI, a blank node or a literal.

The RDF graph model is used as a basis for uniformly expressing information across different fields and domains. Figure 2.2 shows an example graph containing information on the city of Berlin.

³complements the URI standard, <http://tools.ietf.org/html/rfc3987> (June 19, 2015)

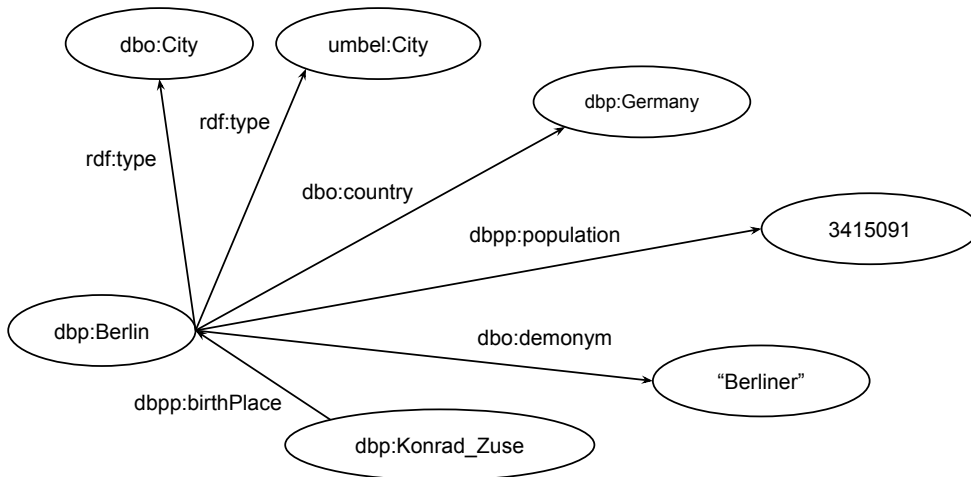


Figure 2.2: An example RDF graph describing the resource with the IRI <http://dbpedia.org/resource/Berlin>, its properties and relationships.

Multiple RDF graphs may be grouped together in an *RDF dataset*⁴. An RDF dataset contains exactly one *default graph* and zero or more *named graphs*. Graphs are named by either an IRI or a blank node. In such cases, statements are also represented as *quads* (4-tuples) where the fourth, additional component denotes the graph name.

A variety of concrete *RDF syntaxes* exists that allow RDF graphs and datasets to be serialized into *RDF documents* so they can be exchanged between systems. An example of the line-based N-Triples⁵ syntax is given in figure 2.3.

By providing a common language for information representation, RDF is one of the core foundations of linked data. The specific properties of the RDF data model have to be taken into account when developing a versioning system for linked data.

⁴since RDF 1.1, <http://www.w3.org/TR/rdf11-datasets/> (June 19, 2015)

⁵<http://www.w3.org/TR/n-triples/> (June 19, 2015)

Background

```
<dbp:Berlin> <rdf:type> <dbo:City> .  
<dbp:Berlin> <rdf:type> <umbel:City> .  
<dbp:Berlin> <dbo:country> <dbp:Germany> .  
<dbp:Berlin> <dbpp:population> "3415091"^^<xsd:integer> .  
<dbp:Berlin> <dbo:demonym> "Berliner"@en .  
<dbp:Konrad_Zuse> <dbpp:birthPlace> <dbp:Berlin> .
```

Figure 2.3: N-Triples serialization of the RDF graph in figure 2.2. Each line encodes a single subject-predicate-object statement from the graph. For readability the prefixes defined in the preliminaries section are used. Usually an N-Triples file would include the full URLs.

3 Motivation

Linked data originates from a variety of data sources. This chapter presents common linked data management strategies and how snapshots of semantic web resources are typically provided today.

3.1 Linked Data Publishing Workflows

Linked data specifies conventions for publishing information in a way which allows for uniform access and interpretation. By encoding information according to the RDF specification, linked data achieves interoperability between different data sources.

Typical sources of linked data are [5]:

- RDF *files* served by a web server,
- RDF *embedded* into other document formats on the web,
- linked data views on top of *databases*,
- linked data interfaces to RDF datasets stored in *triplestores*, and
- linked data wrappers around existing *application or web APIs*.

Each of these strategies follows a different process for publishing linked data.

When working with a relatively small set of entities, a plain, file-oriented approach offers a low-overhead solution. RDF files may be edited manually with a text editor or exported from a software tool. The resulting files are then published by uploading them to a web server which serves them under a certain URI and with the appropriate content-type etc. This approach is often used to

publish personal FOAF profiles and also RDF vocabularies.

Instead of creating RDF-only documents, RDF descriptions may also be embedded into other document formats on the web. This way publishers avoid having to maintain separate resources for a machine-processable RDF description and a graphical representation intended for human users. Embedded RDFa¹ for instance may reside in static HTML files or dynamic pages generated by a content-management system.

Another approach of publishing linked data is to define linked data views on top of databases. A number of mapping languages and tools are available [6]. The D2R server² for example allows clients to access the content of relational databases as linked data.

Similarly, RDF datasets maintained inside triplestores are often exposed through linked data interfaces. Triplestores are purpose-built RDF databases which have their own query language called SPARQL (see section 2.1). Linked data interfaces like the popular Pubby³ construct linked data responses by querying a SPARQL endpoint and generating RDF descriptions according to a configurable mapping scheme.

Lastly, wrappers around existing application or web APIs can transform data in a linked data-compatible manner. Virtuoso Sponger⁴ is an example of a middleware system that generates linked data from a variety of sources.

Linked data provides an abstraction which hides the differences between all kinds of information sources and creates a common interface to the data. In most cases, the published data represents the current state of affairs and is continuously updated.

¹<http://www.w3.org/TR/rdfa-core/> (June 22, 2015)

²<http://d2rq.org/d2r-server> (June 22, 2015)

³<http://wifo5-03.informatik.uni-mannheim.de/pubby/> (June 22, 2015)

⁴<http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/VirtSponger>
(June 22, 2015)

3.2 The Need for Linked Data Versioning

As data changes over time, it becomes desirable to preserve past states in order to refer to specific revisions, compare them or assess data quality in terms of temporal criteria.

RDF vocabularies, for instance, establish common concepts for talking about certain domains. Publishers of linked data choose suitable terms from vocabularies, i.e. properties or classes, to express information in a way which allows others to understand it and relate it to information published elsewhere. A variety of vocabularies exists and mappings between the concepts of different vocabularies clarify their relationships.

Like other software, vocabularies are subject to change and it may therefore be necessary to refer to a specific vocabulary version in a dataset, another vocabulary or a mapping.

Explicit version references are also useful to link to a description of a resource at a certain time and provide metadata. From a data engineering perspective, version references allow us to annotate exact resource states to mark inconsistencies or errors, or when proposing patches [7].

They are further helpful when comparing descriptions of the same resource at different points in time. For example, we may want to make statements about the European Union and its member states before and after May 2004 in linked data, referring to two selected snapshots of the corresponding DBpedia resource http://dbpedia.org/resource/European_Union.

Today, a vast amount of facts on our changing world is captured as linked data: Current population numbers, weather data, recent data on financial markets, politics, the environment and more. The constantly updated information is very valuable, but additional relevant insights are often derived from an understanding of the underlying processes (e.g. population growth, climate change etc.).

Being able to step through past versions of linked data resource descriptions promises a more meaningful investigation of the development of such indicators and possible correlations.

Finally, access to a resource's history allows for a better characterization of the resource description itself. Applications can look at how recent the given information is, estimate its change frequency and determine when to check back for updates. If the data has not been updated in a long time, clients may choose to re-evaluate its trustworthiness and prefer other information sources instead.

Unfortunately, snapshots of linked data are usually not taken and published in a way that makes them automatically discoverable and accessible.

3.3 Versioning Practice

Looking at the sources of linked data, a range of versioning strategies have emerged. While versioning is not mandated by the linked data approach, many dataset maintainers recognize the value of regular snapshots for recovery purposes and retrospection.

When working with versioned RDF files, changes are typically tracked using a *version control system* (VCS) like CVS, Subversion, Mercurial or Git. These are well-established tools in software engineering, mainly for versioning text files, in particular source code. Given a suitable text-based RDF serialization format, they may equally well be used for versioning RDF data.

Preferably, a chosen RDF syntax is line-oriented and guarantees stable ordering to accommodate how version control systems process files. Otherwise, due to the sequential nature of files in contrast to RDF set semantics, the performance of the versioning system may suffer significantly. When no further ordering constraints are placed on the employed serialization algorithm, diffs can become as large as the data itself even for equivalent graphs [8]. Storage will not be as compact, retrieval not as efficient.

While practicable for small sets of files, the VCS-approach also has limitations due to file-orientation and VCS characteristics. It generally does not work well with very large files and for large numbers of files because the costs of creating new resource revisions increase.

Even more, the central issues of version discovery and access remain. Some

version control systems provide web interfaces for inspecting past states of files remotely. However, the relationship between the current resource state and past versions is not clearly stated. Looking at the published linked data, previously available states are not explicitly identified and referenced.

On the other hand, data maintained inside databases and triplestores is typically modified directly and overwritten or removed information is not retained. Data manipulation language statements, e.g. in SQL or SPARQL Update⁵, alter the internal state of the data store and cannot always be reversed.

Probably the most common versioning strategy in such cases are occasional database or RDF dumps, published according to some naming scheme. Usually a version identifier or datetime is encoded in the file path.

These dumps are commonly monolithic exports of the full dataset resulting in large downloads even if clients are only interested in individual resources. Especially when working with multiple datasets this creates a huge overhead for version retrieval and the whole point of *linked* data is connecting different data sources.

Additionally, there is no way of telling how revisions are identified and how they relate to each other without knowing the used naming conventions. These conventions in turn vary greatly across datasets.

For an illustration of this fact consider two real-world examples of published RDF dumps:

1. <ftp://ftp.ebi.ac.uk/pub/databases/RDF/biomodels/r26/biomodels-rdf.tar.bz2>, a release of BioModels⁶, and
2. <http://tools.wmflabs.org/wikidata-exports/rdf/exports/20150511/wikidata-terms.nt.gz>, an export from Wikidata⁷.

Here, `r26` denotes a release number while `20150511` encodes a date. Dates can be encoded in various ambiguous formats. With sequential release identifiers it

⁵<http://www.w3.org/TR/sparql11-update/> (June 19, 2015)

⁶<https://www.ebi.ac.uk/rdf/documentation/biomodels> (June 23, 2015)

⁷<http://tools.wmflabs.org/wikidata-exports/rdf/> (June 23, 2015)

is not apparent, when a certain version was actually created.

The former dump is not even accessible via HTTP and its URI points to a bziped tar bundle containing various RDF/XML⁸ and Turtle⁹ files while the latter points to a single gzipped N-Triples file.

Access to the contained version information in dumps is not consistent like it should be in the linked data world. The main reason being, that such dumps are primarily intended for interested humans, not programs. You usually have to follow a series of links on hypertext pages to find them.

Similar considerations are true for linked data wrappers for application and web APIs. They usually generate linked data based on the current state of the underlying data source and are mostly not version-aware. If snapshots are taken from wrapper-generated resources, they are typically published in the form of dumps, e.g. web crawlers and monitoring tools often channel results into compressed archives.

All in all, today's linked data versioning practice does not make snapshots of past resource states uniformly discoverable and accessible, drastically limiting their usefulness. Some common versioning approaches do not allow the reconstruction of prior states of linked data resources with feasible effort.

The amount of work required to find, set up and maintain more advanced versioning solutions presents a serious entry barrier for many dataset publishers. This thesis explores the idea of a linked data versioning service and how it could improve this situation.

⁸<http://www.w3.org/TR/rdf-syntax-grammar/> (June 23, 2015)

⁹<http://www.w3.org/TR/turtle/> (June 23, 2015)

4 Related Work

This chapter discusses related work on versioning in the context of linked data.

4.1 RDF Change Descriptions

We have already discussed RDF syntaxes in section 2.2. A simple strategy for preserving dataset history is to make full copies of such descriptions and archive them. This, however, requires a lot of storage space for data which continuously and gradually evolves.

Another way of preserving dataset history is to describe data changes via RDF using specialized vocabularies or through patch formats. A published series of such change descriptions would theoretically allow clients to reconstruct prior resource states and could be more compact than full copies.

4.1.1 Change Vocabularies and Ontologies

A variety of vocabularies and ontologies allow for describing changes to RDF datasets. They represent update information by means of RDF itself.

*Changeset*¹ defines a vocabulary for changes to resource descriptions using RDF reification: An update is represented by a set of statements about statements and whether they are added or removed. Additionally, the vocabulary includes terms for supplying metadata on the change, in particular the date it was created and the preceding changeset.

¹<http://vocab.org/changeset/schema.html> (June 23, 2015)

Gutierrez et al. [9] have proposed a temporal extension of RDF graphs for asserting when a contained statement is considered valid. Their *temporal* vocabulary uses reification for time-labelling each individual RDF statement in a graph.

The *Graph Update Ontology* (GUO)² provides an OWL³ ontology for describing graph changes which avoids RDF reification but requires a very different interpretation. With GUO, updates are encoded as graph nodes which have properties indicating the actual target node of the update operation. These GUO nodes further reference artificial blank nodes connecting the properties and objects to add or remove to the target node.

The *log* ontology [10] was developed to describe high-level changes to aid in ontological engineering. It was integrated into the collaborative RDF editor Powl⁴ which is meanwhile discontinued.

A problem with using RDF vocabularies or OWL ontologies for versioning purposes is that the change representations they yield are generally not very compact. For each atomic change, a multitude of new RDF statements is created. Ideas to encode changes diverge and tools to actually apply the described updates are missing. None of these vocabularies has developed into an accepted standard and the publication of change descriptions in RDF is not commonly adopted.

4.1.2 Patch Formats

Patch formats are basically the syntactical counterparts to change vocabularies and ontologies. Both *Delta* [8] and *RDF Patch*⁵ propose dedicated patch file formats to represent differences between RDF models for change inspection and propagation between systems.

While not as expressive as RDF, they are mostly derived from RDF syntaxes which facilitates patch generation and processing. Patch formats tend to be more compact than the semantic change descriptions presented in the previous

²<http://purl.org/hpi/guo#> (June 23, 2015)

³<http://www.w3.org/TR/owl-features/> (June 23, 2015)

⁴<http://aksw.org/Projects/Powl.html> (June 23, 2015)

⁵<http://afs.github.io/rdf-patch/> (June 23, 2015)

section, making them good candidates for many practical and space-conscious applications.

4.2 HTTP-based Version Access

Apart from finding a way of representing version information, it is also necessary to provide access to stored revisions. A time travel protocol for the web has been previously proposed by the *Memento Project*⁶ [11], which aims at making archived web resources easily accessible via time-based content negotiation. The protocol is natively supported by a number of web archives, such as the Internet Archive⁷ and public libraries. It is available as an extension to MediaWiki instances⁸ and a proxy implementation exists for a range of services which offer their own version API.

The Memento approach has also been adopted in the context of linked data, e.g. to provide access to prior versions of DBpedia resources [12]. To achieve this, DBpedia releases were stored in a MySQL database as complete snapshots and served through a Memento endpoint (a more detailed description of the protocol follows in section 7.2). The demonstrator, however, misses an interface for creating new revisions and updates have been discontinued with DBpedia 3.9.

Nevertheless, the Memento protocol offers a version discovery and access mechanism that aligns well with linked data principles. It uses the HTTP standard to communicate links to prior resource states. Clients can easily discover whether a resource supports versioning and where to retrieve resource snapshots.

Furthermore, Memento establishes a mechanism for time-based querying. A client does not need to download and interpret a meta description for the resource history before retrieving a snapshot. Instead, it specifies the date and time it is interested in and lets the server handle revision reconstruction. Whatever the internal representation of resource history, the client has no need of knowing used conventions, perform ontology mapping or other complex trans-

⁶<http://mementoweb.org> (June 23, 2015)

⁷<https://archive.org/> (June 23, 2015)

⁸<http://www.mediawiki.org/wiki/Extension:Memento> (June 23, 2015)

formations .

This kind of separation also offers flexibility in evolving the used revision storage model without requiring additional effort on the client-side.

4.3 RDF Store Versioning

Large RDF datasets are typically maintained inside specialized RDF stores. Several projects have proposed the development of RDF stores with versioning capabilities.

4.3.1 Version Control for RDF Triple Stores

Cassidy et al. [13] have proposed a version control system for RDF inspired by Darcs⁹ and its semi-formal theory of patches. Their goal is to apply source code management strategies to the development of RDF datasets.

The system they describe, like Darcs, is entirely built around patches. Patches are represented as named graphs containing separate nodes for each added and deleted statement in the versioned graph. Each of these nodes has properties for subject, predicate and object as well as the type of the change. Hence, for each single addition or deletion from the RDF store at least four new statements are introduced for the patch.

The authors conclude that their system for version control “is around four to eight times slower and needs from two to four times as much space as the raw RDF store”.

4.3.2 R&Wbase

R&Wbase [14] tracks changes to an RDF dataset within a modified triplestore implementation. Changes to the whole dataset are encoded via context values assigned to the changed triples for each revision. These context values are

⁹<http://darcs.net/> (June 24, 2015)

numbers from a continuous sequence: Additions are marked by even values, deletions receive odd values (essentially the least significant bit distinguishes addition from deletion). In order to restore a revision, triples are scanned and selected if the highest context value found for a combination of subject, predicate and object is even.

R&Wbase allows querying the dataset via SPARQL and uses virtual graphs for revision access.

It envisions a Git-like tool geared towards distributed dataset development and reimplements traditional source code management strategies. The approach supports advanced versioning concepts like branching and merging to form commit graphs with multiple lines of development.

The proof-of-concept implementation is based on an outdated version of the Virtuoso¹⁰ triplestore and is no longer actively maintained. Its branching model incurs higher overhead for version reconstruction than needed for single-history archival.

Finally, while R&Wbase proposes a workflow for the development of a single RDF dataset, it does not consider the linked data aspect of the managed information.

4.3.3 R43ples

R43ples [15] implements an RDF-versioning proxy in front of a SPARQL endpoint. It uses named RDF graphs to group additions and deletions and rewrites client requests in order to implement version queries and updates. R43ples performs version control on a graph level.

Much like R&Wbase, R43ples aims at being a development tool with additional semantics for tagging, branching and merging. It introduces its own set of non-standard SPARQL keywords for version control commands and querying past states of the data.

Revisions of the data are restored by creating a temporary graph from the head

¹⁰<https://github.com/openlink/virtuoso-opensource> (June 24, 2015)

revision, then rolling back changes stored in the preceding addition and deletion graphs.

The temporary copies created for graph diffing and revision reconstruction turn out to be rather costly. The comparably poor performance of the approach limits its use to “medium-sized datasets” with short histories.

4.3.4 Apache Marmotta

KiWi, which is now part of the Apache Marmotta project, implements a triplestore backend on top of a relational database, i.e. H2, PostgreSQL or MySQL. An optional versioning module¹¹ for KiWi can be compiled into the Java project to enable support for change tracking.

The versioning module uses *tuple versioning*, assigning `created` and `deleted` timestamps to every row in the underlying database table. Instead of physically deleting statement information, the deletion timestamp is set to the current date and time which allows them to be filtered at a later point. Past states of the triplestore content can be inspected through a Memento interface in Marmotta.

Unfortunately, versioning support in Marmotta is apparently not widely used. In their 2014 paper, Embury et al. [16] report that they “couldn’t find any publicly accessible versioned data sources that used Marmotta”.

The overall data model is designed to be used within one organization, with a group of trusted users operating on a single, shared dataset.

4.4 Versioning Services

A few versioning services exist on the web, both for common web pages and semantic web resources in particular.

The *Wayback Machine*¹² archives a large part of the world wide web through its crawls, including many linked data resources. As such, it has become an

¹¹<http://marmotta.apache.org/kiwi/versioning.html> (June 24, 2015)

¹²<http://waybackmachine.org/> (June 25, 2015)

Related Work

invaluable resource of historic web data. The site's users, however, only have limited control over when snapshots of certain resources are actually taken.

*Linked Open Vocabularies (LOV)*¹³ focuses entirely on a small part of the semantic web: It hosts meta information on a curated list of RDF vocabularies. Users can suggest vocabularies which are then reviewed by a team [17]. If a vocabulary is accepted, it will then be monitored on a daily basis to detect updates. Some past versions of selected vocabularies are available for download.

¹³<http://lov.okfn.org/dataset/lov> (June 25, 2015)

5 Approach

We have seen, in the previous chapter, that a number of approaches to preserving dataset history have been proposed, in particular change descriptions and RDF store versioning. However, none of these solutions has yet seen widespread adoption as versions are typically not stored together with the original data. If snapshots of linked data sources are taken, they are usually not created and published in a way which allows linked data clients to discover and access them.

For instance, when working on a small dataset in the form of relatively few files, the overhead for operating a versioning RDF store instance is daunting for most developers. Simpler tools are preferred for version control in the development process, i.e. the RDF files are updated locally, maybe committed to a VCS repository and then copied to a web server where they replace prior versions. Usually there is no visible connection between the last, published version and its archived states.

Even larger projects using triplestores internally to manage linked data have not yet, to a large extent, adopted RDF store versioning solutions. Instead, we see that they stick to publishing RDF dumps placed on static file servers. Mostly, these are referenced through hypertext links on HTML pages for other developers to download them if they are interested, but not for linked data applications.

In parallel, there is extensive work on linked data crawling and monitoring tools in the research community [18, 19, 20]. The *Dynamic Linked Data Observatory* (DyLDO) [21] performs weekly crawls on a subset of the web of data and gathers a huge amount of valuable snapshots of linked data resources. Unfortunately, these are then also published as dumps and hence not accessible by linked data clients.

Ideally, what we would like to achieve is to create a storage platform for versions of linked data resources, to which publishers (or observers) can easily push resource snapshots and reference them so they are discoverable on the linked data web.

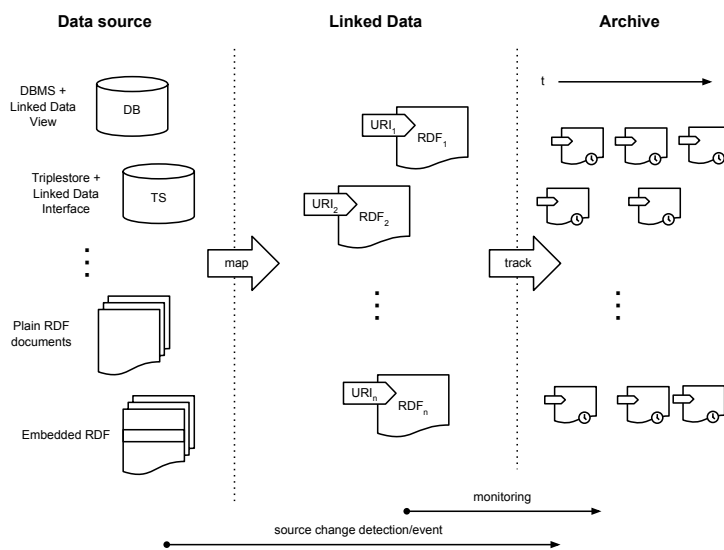


Figure 5.1: Linked datasets originate from a variety of different data sources. A versioning system receives notifications about changes to a dataset directly from the publisher or through monitoring tools.

Different sources for linked data exist, e.g. views defined on top of relational databases or dedicated triplestores, data published in plain RDF documents or RDF embedded in other document formats like HTML (figure 5.1).

By approaching the problem of recording dataset history on a linked data level, we achieve an abstraction over different sources of data and are able to establish a consistent interface to historic states.

5.1 A Model for Linked Datasets

While RDF concepts have been formalized to a large degree, the linked data principles are stated in natural language. To gain a common understanding of our interpretation of the linked data principles, we will give a brief definition of the core ideas.

Following the linked data principles, a *linked data resource* is identified by an HTTP URI and described via RDF. We mathematically represent a resource and its associated RDF description as a pair (URI, RDF) . The *URI* component is the address that can be used to refer to this resource and to retrieve its *RDF* metadata description.

Note that when we say *RDF description*, we refer to the abstract model for the resource, not a concrete serialization in an RDF syntax. Technically, retrieving the content of a web resource may yield RDF descriptions in different formats.

Now, a *linked dataset* is simply a set of linked data resources along with their descriptions, a binary relation

$$DS = \{(URI_1, RDF_1), \dots, (URI_n, RDF_n)\}$$

in which each URI can appear at most once (ger. *rechtseindeutige binäre Relation*). By definition, this relation can be equally described as a partial function from the set of valid HTTP URIs to the set of possible RDF descriptions.

Adding the notion of time to linked datasets, a dataset may undergo changes and evolve at certain points in time $t_0 < \dots < t_k$:

$$\begin{aligned} DS_{t_0} &= \{(URI_{t_0,1}, RDF_{t_0,1}), \dots, (URI_{t_0,n_0}, RDF_{t_0,n_0})\} \\ DS_{t_1} &= \{(URI_{t_1,1}, RDF_{t_1,1}), \dots, (URI_{t_1,n_1}, RDF_{t_1,n_1})\} \\ &\vdots \\ DS_{t_k} &= \{(URI_{t_k,1}, RDF_{t_k,1}), \dots, (URI_{t_k,n_k}, RDF_{t_k,n_k})\} \end{aligned}$$

in which case we refer to $DS_{t_0}, \dots, DS_{t_k}$ as *dataset states* and $RDF_{t_0,1}, \dots, RDF_{t_k,n_k}$

as *resource states* (as an identifier, the URI for a resource never changes and thus resource state is encoded in the RDF description¹).

5.2 Possible Changes in Linked Datasets

Based on this model view, we can identify these possible (atomic) changes between subsequent dataset states:

- The introduction of a URI-RDF description pair.
- The removal of a URI-RDF description pair.
- Changes to the RDF description associated with a URI.

Typically, such changes are not recorded. The data source is modified in place and only the current state of the data is kept.

5.3 Querying Linked Dataset History

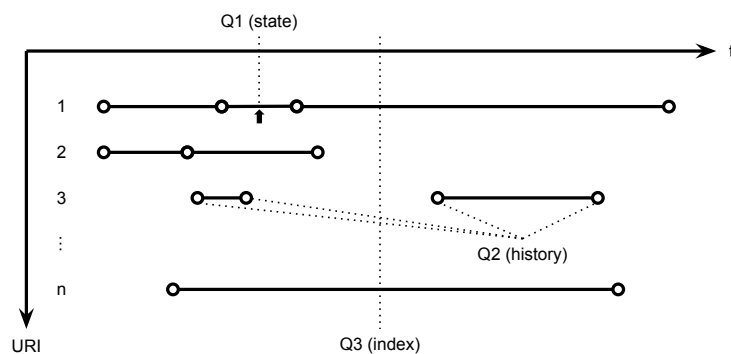


Figure 5.2: Linked dataset history consisting of individual timelines for each resource and a visualization of the basic types of queries.

A linked data archival system should answer the following essential queries with regard to dataset history (in order of importance):

¹<http://www.w3.org/TR/webarch/>, section 3.5.1 “URI persistence” (June 28, 2015)

Q1 What was the description RDF_x of URI_x at time t ?

Q2 When did the description for URI_x change?

Q3 Which URI_* existed at time t ?

A graphical illustration of these queries is given in figure 5.2.

Q1 allows the retrieval of a past state of a specific resource, a static snapshot or “memento” [22]. We consider this to be the most common type of query.

A client will use such a query to request a specific resource state referenced in another linked data description or to retrieve a certain state as part of iterating the resource history.

Q2 allows for tracing changes to a single resource through time, figuring out at what points in time it was actually changed.

Queries of this kind could be used to analyze temporal change characteristics, for instance, the change frequency of a resource or how recently it has been modified. In combination with Q1 queries, it can be used to further investigate content changes throughout the resource’s history, e.g. what fraction of statements is changed or which statements in particular.

Q3 asks for an index of the resources represented in a dataset at a certain point in time.

Finding out about all resources in a dataset may be used as a starting point for other queries. However, we suspect that, usually, time-based investigation will start with a specific resource, i.e. requesting it directly or through a reference.

Other, high-level queries like cross-version queries may be implemented in terms of these basic classes of queries.

6 Investigated Alternatives

Before arriving at our final system design, we have experimented with different implementation approaches for our linked data versioning service. This chapter describes attempted realization paths and issues we faced.

6.1 Triplestore Extension

An approach we investigated was extending an existing triplestore implementation to support versioning and time-based retrieval of linked data resources.

6.1.1 Native Triplestores

Early triplestore implementations were based on relational databases. More and more projects however move towards native triplestore implementations. Two well-known and mature open-source projects have deprecated their RDBMS-based stores and now recommend the newer purpose-built triplestore variants:

- *Apache Jena*¹ is moving from *SDB*, based on a relational database backend, to the native *TDB*, promising better speed, scalability and support.
- *Sesame*² deprecated its MySQL and PostgreSQL stores in favor of the new, maintained *Sesame Native Store*.

Both of these native triplestore implementations, like others we have found, do not support versioning.

¹<https://jena.apache.org/> (June 30, 2015)

²<http://rdf4j.org/> (June 30, 2015)

We refrained from a modification of a native triplestore with regard to breaking changes in future updates to the original source code. Integrating versioning support and support for multiple datasets would require widespread changes in many of the triplestore components (data structures, persistence layer, query engine and so forth). Given that these projects are still under active development, it appears unlikely a heavily modified variant could benefit from mainline improvements without conflicts.

6.1.2 Virtuoso Universal Server

Inspired by R&Wbase [14], we looked into the possibility of adapting an up-to-date *Virtuoso Universal Server*³ (version 7).

Virtuoso is a middleware and database engine hybrid which incorporates the functionality of different data store and server classes. It supports relational databases but is also widely used as a triplestore.

Virtuoso manages RDF data through its own relational database engine. A large triple/quad table is used to store a single RDF dataset and additional tables store namespaces, URI-mappings and blank node identifiers. To efficiently work with RDF data, Virtuoso has built-in support for RDF datatypes and procedures for handling RDF terms.

The hybrid RDBMS-RDF store character of Virtuoso looked as if it would enable us to more easily alter its quad table schema, introduce additional time and user information. The many custom RDF extensions however made Virtuoso hard to adapt for our purpose of implementing a linked data versioning service.

We had to identify the functions Virtuoso uses internally for handling URI, blank node and literal encoding and bulk insertion of statements. These are natively implemented in C and modify the RDF store directly.

Using them on our modified storage schema caused unwanted side-effects. RDF store transactions altered rows from a previous insertion and corrupted the data because the internal implementation of those functions relied on the conven-

³<http://virtuoso.openlinksw.com/> (June 30, 2015)

tional quad table schema.

6.2 VCS-based Implementation

Another path we investigated was basing our service implementation on a regular version control system. Such systems are primarily intended for managing software source code. Using a textual RDF representation they may be used to version linked datasets as sets of RDF files as well.

Version control systems usually process files in a line-oriented, sequential manner, i.e. for diffing. To accommodate for this, the versioned RDF files should be created using an RDF syntax which encodes one statement per line and applies additional ordering constraints, e.g. sorted N-Triples or N-Quads⁴.

We have selected two popular version control systems, Git⁵ and Mercurial⁶ for our experiments. Both support similar versioning functionality, but implement different storage models. We were interested in whether one of these systems could be used to efficiently version linked datasets.

6.2.1 Git

Git is essentially built on the concept of a content-addressable object store which identifies and retrieves objects based on a checksum of their content [23]. Object data is stored on the underlying filesystem in a directory structure where the file path is derived from the content hash, e.g. an object which hashes to `d6cef2b706db6ca52192829103473c8dd99919ef` is stored in a file named `.git/objects/d6/cef2b706db6ca52192829103473c8dd99919ef`.

Git distinguishes three types of objects to represent repository history:

- *commits*,
- *trees* and

⁴<http://www.w3.org/TR/n-quads/> (June 30, 2015)

⁵<http://git-scm.com/> (June 24, 2015)

⁶<https://mercurial.selenic.com/> (June 24, 2015)

- *blobs*.

Commit objects form the backbone of the repository history. Each commit contains a pointer to the commit which precedes it. They thereby form chains allowing to retrace any changes to the repository. Additionally, commits store meta information on the change, i.e. the time the commit was made, its author and a commit message. Finally, a commit points to a tree object for the stored state of the repository root directory. Trees represent the state of a directory within a repository. They point to other tree objects for subdirectories, or blobs. A blob object encapsulates the contents of a versioned file at a certain point in time, a complete snapshot. All objects are stored according to the same filesystem structure described above.

In an early experiment, we have evaluated Git on a sample of DBpedia resources. A bare Git repository was initialized and the sample resources from DBpedia release 3.8 were added, one file per resource description. Afterwards, a new version for each of the resource files was committed with the content updated from DBpedia 3.9.

Instead of using Git's user-oriented "porcelain" commands, we created Git objects directly to remove the need for a working copy and reduce unnecessary filesystem accesses (`lstat(2)`⁷ calls to determine which files have changed and file content reads).

Git regularly stores objects in *loose* format, meaning each object is stored in a separate file. Object content is `zlib`⁸-compressed but file content is not stored as a delta against previous versions. Even if you only change a single line in a large file, Git stores the new content as a completely new object. Git however has a *garbage collection* (`gc`) operation which will pack loose objects into so-called *packfiles*, look for similar objects and delta-compress them.

During our test run, we periodically triggered garbage collection every 100 commits to keep variations in repository size within an acceptable limit. You can see the characteristic sawtooth curve for the oscillating repository size in figure 6.1 (only the first 5,000 commits are shown). There is a steep growth in the reposi-

⁷<http://linux.die.net/man/2/lstat> (June 30, 2015)

⁸<http://www.zlib.net/> (June 30, 2015)

tory size as each update creates new loose commit, tree and blob objects. After 100 commits, repository size is reduced again by repacking.

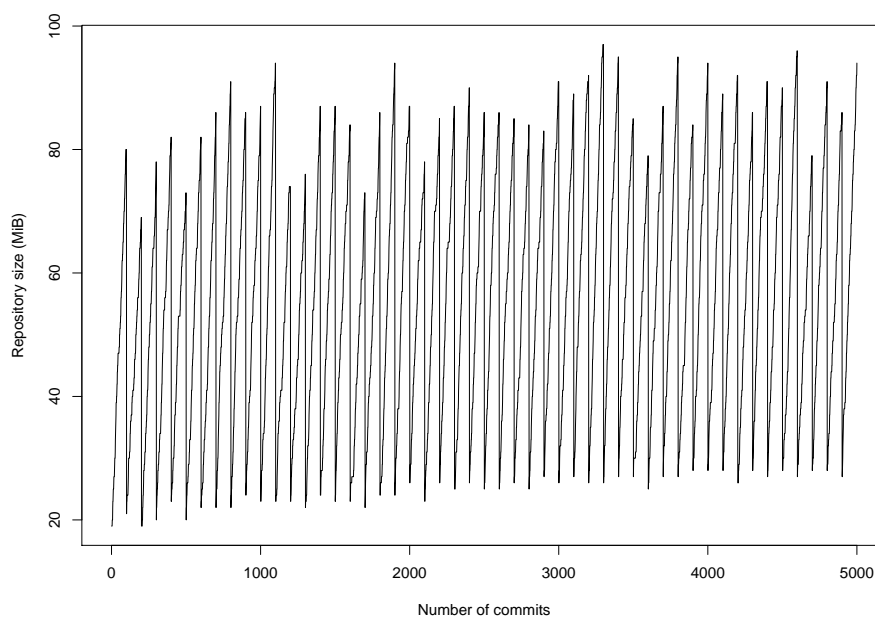


Figure 6.1: Git repository size for first 5,000 commits of the experiment, measured after each commit. Size is periodically reduced by triggering garbage collection every 100 commits.

After all, while commits were generally fast, a limiting factor for this approach was gc time. Figure 6.2 visualizes the time taken for the first 50 gc invocations (across 5,000 commits) during our experiment. Measured gc time exhibited linear growth behaviour with regard to the number of recorded updates (updates were similar in size) and continued to increase in this way until we aborted the test run after more than 13,700 commits at which point each gc invocation already took more than one minute.

For large, changing datasets, such variations in repository size and growth in compaction time appear impracticable for a service implementation.

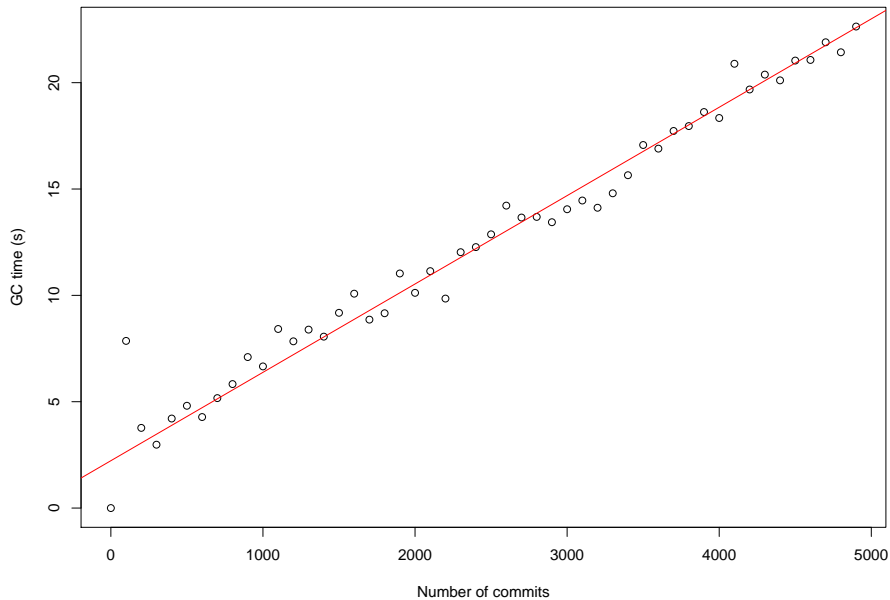


Figure 6.2: Git garbage collection time in relation to the number of commits made (number of updated resource descriptions).

6.2.2 Mercurial

In contrast to Git’s snapshots-first storage design, Mercurial implements a storage model which more commonly stores deltas between subsequent versions. One can therefore expect a similarly compact repository size, but a more gradual increase.

The core structures Mercurial uses to store repository history are called [24]:

- *changelog*,
- *manifest* and
- *filelog*.

In Mercurial, the changelog contains information about all changes to the repository. Each revision of the changelog identifies the author of the commit, when

it was made etc. Just like each Git commit points to a tree object, a changelog revision specifies the version of the manifest to use. The manifest, in turn, records which files are present in that version of the project and what revision of the file is valid. Finally, each individual file's history is stored in a filelog.

All of these metadata structures are stored in the same format, a *revlog*. A revlog encodes revisions of a file as deltas on top of a snapshot. Specific file revisions are recreated by reading the latest snapshot and subsequent deltas. To limit revision retrieval time, Mercurial departs from the delta-only approach and creates new snapshots when the accumulated size of deltas since the last snapshot exceeds a fixed threshold.

To test how Mercurial performs as a version control system for RDF files, we used the same process we used to test Git. Note that we had difficulties with committing changes to a bare Mercurial repository. The measured repository size shown in figure 6.3 excludes the size of the working copy.

The initial on-disk size for the Mercurial repository is much higher compared to that of Git. This is due to Mercurial allocating a separate filelog for each individual resource and a per-file overhead introduced by the fixed filesystem block size. Git is not affected by this overhead as much, because of packing.

While the initial values for our experiments with Git and Mercurial differ significantly, it is apparent that the growth in repository size is less steep for Mercurial.

Mercurial appends mostly deltas to existing changelog, manifest and filelog objects instead of creating completely new snapshots. This means that repositories tracking a fixed set of files with gradual changes will grow more slowly, even without a Git-like garbage collection operation. The initial size difference will diminish as soon as additional revisions of existing files are added.

The reason we chose not to continue with Mercurial is: Its implementation assumes that a single file, index or manifest can be read into system memory in its entirety⁹. While this assumption is probably valid for most source code projects, it fails for projects with a lot of files (because the manifest will be large and it is reconstructed in memory) or large files (e.g. if you try to group multiple resource

⁹<https://mercurial.selenic.com/wiki/FAQ/TechnicalDetails> (June 30, 2015)

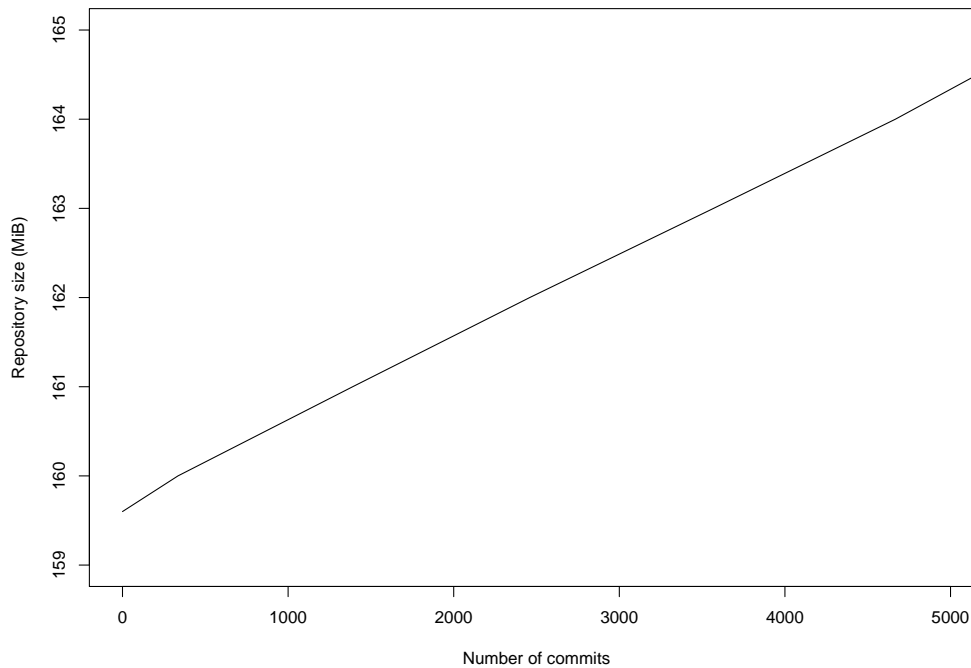


Figure 6.3: Mercurial repository size for the first 5,000 commits of the experiment, measured after each commit. Size increases more gradually compared to Git with no garbage collection required.

descriptions into a single file). During an experiment with a larger dataset, Mercurial eventually filled up the test machine’s memory and caused the process to exit with an error.

Overall, the high memory consumption and costly reconstruction of manifests for resource retrieval is not well suited for the implementation of a linked data versioning service.

6.3 Summary

Even though these experiments did not yield a strong candidate for our service implementation, we think that these results are also valuable for others who are

interested in (linked) data versioning.

Unfortunately, all the open triplestore implementations we have found lacked support for versioning and/or multiple datasets. Extending a native, open-source triplestore with such capabilities would require modifications to many interdependent system parts, which is why we chose not to pursue this approach.

For a service implementation that deals with many datasets simultaneously, we needed a storage design which would grow predictably and which does not require fitting large files into memory. The version control systems we evaluated, Git and Mercurial, did not fulfil these criteria. It is likely, that the same applies to other VCS implementations.

In the end, our service implementation could not use any of the investigated approaches directly, but we adapted some of the abstract ideas found in these systems.

There are few more options which could be further explored and compared as future work. We will give a brief discussion in chapter 9.

7 Implementation

This chapter presents our implementation of a linked data versioning service. The described interfaces and storage model have been implemented as a Python web service on top of a relational database management system.

7.1 System Overview

The platform we created aims to provide linked data archival as a user-friendly service. Users or organizations can register freely and create repositories for the linked datasets that they wish to track. Generally, dataset maintainers will want to create a repository for their own dataset, but others may choose to monitor public datasets and submit them to the service if an official repository is missing.

Apart from a graphical web interface, which allows users to manage their accounts and browse repositories, the service comprises two major HTTP APIs:

- A *Push API* for submitting dataset change information to the system.
- A read-only *Memento API* for accessing versions of stored linked data resources.

The following sections describe these system APIs and the underlying storage model for the current system.

7.2 Memento API

The idea of a time-travel interface for the web has previously been developed and advocated by Van de Sompel et al. [11]. Their protocol-based framework,

called “Memento”, uses standard HTTP capabilities in order to link and retrieve past states of web resources¹. The archival platform we propose applies the concepts of Memento to linked data resources for consistent and discoverable version access.

To begin with, the Memento approach distinguishes four types of web resources:

original The original resource is a resource that exists or used to exist, and for which the system provides access to prior states.

timegate The timegate is the resource providing access to prior states of the original using datetime negotiation.

memento A memento is a resource representing a prior state of the original resource at a certain point in time.

timemap The timemap is a resource which lists URIs and time information of the mementoes available for an original resource.

A major advantage of this distinction is that it allows for different systems to handle requests for each resource type, i.e. original resources and past states may be served by separate hosts.

Original

In our case, the *original* resources, for which we want to provide access to prior states, are arbitrary linked data resources published on the web.

Timegate

Each repository created with our service acts as a *timegate* for such original resources. It supports datetime negotiation for accessing prior resource states, the *mementoes*. The datetime negotiation mechanism works very similar to regular HTTP content negotiation. Except instead of a document format, a certain point in time is requested by passing an `Accept-Datetime` header (Q1-type query, section 5.3).

¹<http://tools.ietf.org/html/rfc7089> (July 2, 2015)

Implementation

For example, entities from the *Upper Mapping and Binding Exchange Layer* (UMBEL)² might be tracked in a repository `umbel/entities`. If a client was interested in a past state of UMBEL's `PlanetPluto` resource, it could request

```
/api/umbel/entities/http://umbel.org/umbel/rc/PlanetPluto
```

from the service, passing the `HTTP Accept-Datetime` header with an RFC 2616³-compliant datetime value, e.g. `Sun, 15 Aug 2010 02:14:00 GMT`.

Mementoes

To a request like the one above, the server will reply with an RDF description equivalent to that of the tracked original resource valid at the given point in time. The response will also include a `Memento-Datetime` header informing the client about when the resource was actually changed. This might be earlier than the actual requested timestamp. In case the resource did not yet exist at that time or was deleted previously, the server responds with the appropriate HTTP 404 status code.

For straight-forward linking to a specific version of a resource, it is also possible to specify the desired datetime as a path component instead of a header argument, e.g.:

```
/api/umbel/entities/20100815021400/http://umbel.org/...
```

An index of all the URIs in a dataset at any given time (Q3) may be retrieved using the same datetime negotiation mechanism described above. For datasets consisting of many resources, this index is split up into a number of pages. The service provides this additional, virtual index resource as an extension to the Memento framework.

²<http://umbel.org/> (July 4, 2015)

³<http://www.ietf.org/rfc/rfc2616> (July 4, 2015)

Timemap

If a client is interested in the change history of a particular resource (Q2 query), it can request that resource's *timemap*. A timemap lists links to all of the stored states for a resource along with their timestamps.

The service implementation supports two different timemap serializations: The link format from the initial Memento RFC or the newer JSON timemap⁴, a more common format in modern web applications.

To figure out, when the `PlanetPluto` resource changed and where to find the mementoes for all of its prior states for instance, a client will request:

```
/api/umbel/entities/history/http://umbel.org/umbel/rc/PlanetPluto
```

A response to this request, using the JSON serialization, will return a timemap object of the following form:

```
{
  "original_uri": "http://umbel.org/umbel/rc/PlanetPluto",
  "mementos": {
    "list": [
      {
        "datetime": "2010-08-15T02:14:00Z",
        "uri": "http://tailr.s16a.org/api/umbel/entities/2010
              0815021400/http://umbel.org/umbel/rc/PlanetPluto"
      },
      ...
    ]
  }
}
```

⁴<http://mementoweb.org/guide/timemap-json/> (July 4, 2015)

Linking

The Memento framework makes use of the HTTP `Link` header⁵ for supplying URI references between the different kinds of resources. The service memento responses will always contain the URIs of the corresponding original resource and timemap:

```
Link: <http://umbel.org/umbel/rc/PlanetPluto>; rel="original",  
      <http://tailr.s16a.org/umbel/entities/history/...>; rel="timemap"
```

For best integration, the original linked data resource should refer to the service as its timegate by inserting an appropriate `Link` header on its own. Because the service's timegate, memento and timemap URIs can be easily constructed from the original resource URI, a simple Apache or Nginx proxy configuration change will suffice in most cases to achieve this explicit reference.

Summary

A major advantage in using Memento for a linked data archival platform is that it closely mirrors linked data principles by using HTTP and content-negotiation for version retrieval. It is a well-documented approach which offers great discoverability through explicit references and is easily integrated with existing linked datasets.

Because Memento differentiates between different kinds of resources, a service like ours can provide prior version access for original resources hosted elsewhere.

7.3 Storage Model

The current storage model of our service draws inspiration from concepts found in existing version control systems and adapts some of their ideas.

⁵<https://tools.ietf.org/html/rfc5988> (July 4, 2015)

Conceptual View

There are two common model alternatives in version control systems: *snapshot storage* and *delta storage*. With snapshot storage, full descriptions are stored for each new revision, while a delta approach focuses on encoding transformations that lead from one version of the data to the next.

Both of these models have different trade-offs regarding storage space and revision retrieval time. Retrieval of revisions tends to be most efficient with snapshot storage as there is no computational overhead involved for applying data transformations. In contrast, a delta model is often more compact for data which undergoes evolutionary changes (a small portion of the data changes with each revision).

The system we describe uses hybrid revision storage to balance space requirements and retrieval time (cf. [25]). The history of each individual tracked resource is encoded as a series of deltas based on interspersed snapshots as shown in figure 7.1. A specific revision can be either fetched directly, in case a snapshot exists, or recreated with the help of the preceding snapshot and following deltas.

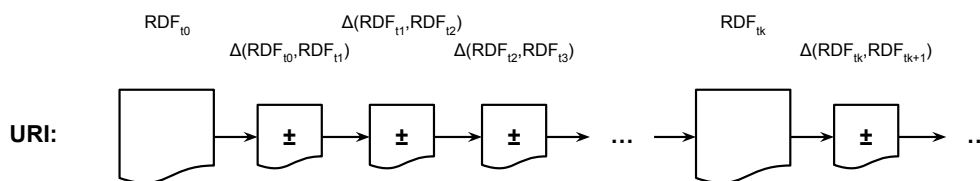


Figure 7.1: Our hybrid storage model balances revision reconstruction costs and space requirements. Resource revisions are typically stored as deltas on top of previous states, but from time to time snapshots are inserted.

Implementation

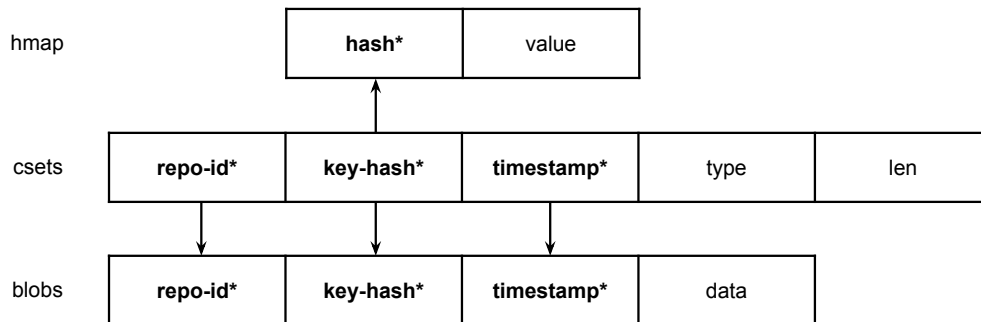


Figure 7.2: Storage schema used to record dataset changes. A * indicates a primary key component.

Model Implementation

The hybrid storage model is currently implemented on top of a relational database in MariaDB⁶. The storage structure is designed to support memento queries. Basic entities in the system are (see also figure 7.2):

repos Repositories are created by users and are typically referenced by name.

Conceptually, each repository encapsulates a linked dataset and its history.

csets Changesets encode information about changes to linked data resources within a repository. There are three types of changeset entries: *snapshot*, *delta* and *delete*. Each cset entry is identified and indexed by repository id, a hash of the original resource URI (the “storage key”) and its timestamp. We are currently using the SHA-1 hash function, also used in popular DVCS systems like Git or Mercurial, for hashing resource URIs. It produces 20-byte hashes but the hash-size could of course be increased when necessary, swapping it with a different function with longer output.

hmap This auxiliary structure maps hashed storage keys back to their unhashed value. It is only needed if support for queries of type Q3 is required (see section 5.3). The URI hashes stored in the hmap are typically much shorter than storing the plain URI as part of each cset redundantly. Also, queries

⁶<https://mariadb.com/> (July 5, 2015)

Q1 and Q2 can be answered directly without consulting the hmap because with a well-known hash function, the storage key can be computed directly and used for querying.

blobs Blobs contain optional data associated with cset entries. In the case of snapshots, they contain a normalized version of the resource RDF description. Blobs for deltas encode differences in the resource description on top of the previous resource state (added and removed statements). No blobs are created for delete-csets as the cset marking a resource as deleted already captures just that information.

The csets for a resource form *base+delta chains* which allow reconstruction of resource states by time. The *base* is a non-delta cset (snapshot or delete) and followed by 0 or more *deltas* according to their timestamps. As a general rule: There are no deltas directly after a delete-base. A delete is always followed by a snapshot (details below).

Reconstructing a particular resource state reads from the latest non-delta cset (and corresponding blob) before the requested time t and applies the subsequent deltas in the chain ordered by time. Queried cset entries for resource reconstruction may be read sequentially from a coherent range.

repo-id*	key-hash*	timestamp*	type	len
			SNAPSHOT	
	...		DELTA	
	...	$\leq t$	DELTA	
	...	$> t$	DELTA	
	...		DELETE	
	...		SNAPSHOT	
	...		DELTA	




Figure 7.3: Base+delta chain: Any resource state is reconstructed from the latest non-delta cset prior to the requested time t and subsequent deltas.

A tuneable chain length limit avoids high retrieval costs for resources with long histories that include many changes, i.e. a snapshot is stored if

1. *The chain length for the resource is 0.* When a history for a resource has not

yet been recorded, we simply insert a snapshot. Hence the first cset entry for any resource is always a snapshot. A delta would only mark every statement as “added”.

2. *The last chain entry is a delete.* Similar to condition 1, when inserting a new RDF description after a delete, all statements would be considered “added”. It is thus more efficient to just create a snapshot cset.
3. *The snapshot is smaller than the delta computed against the last revision.* When a significant portion of the RDF description changes, a delta may be larger than a snapshot. Our system does the right thing in such cases and stores the entry occupying less space.
4. *The accumulated size of deltas from the last stored snapshot exceeds a configurable threshold.* By ensuring that snapshots are created on a regular basis, we effectively prevent base+delta chains from growing arbitrarily long and cap the costs for retrieving random revisions.

With this model we can efficiently store and retrieve the revision histories for sets of resources and thus linked datasets as a whole.

A list of resource URIs existing in a repository at a particular point in time is not stored explicitly, but can be derived by traversing csets in a repository and joining with the hmap entries. Our model is not as efficient for this type of query. Rather it is optimized for queries Q1 and Q2 as we assume these to be the most common use-case. The biggest issue with Q3-type queries is: The list of resource URIs in a dataset may be very long for linked datasets making it infeasible to store and version it directly.

Delta Operations

Snapshot blobs contain RDF as zlib-compressed N-Quads⁷, data for deltas is stored in zlib-compressed RDF-Patch format⁸. Both of these formats are syntactically very similar, making it straight-forward to generate patches based on two

⁷<http://www.w3.org/TR/n-quads/> (July 5, 2015)

⁸<http://afs.github.io/rdf-patch/> (July 5, 2015)

N-Quad representations and, reversely, reconstructing models from an N-Quad snapshot and RDF-Patch deltas.

Computing deltas uses a hash set implementation. Sets of statements can be diffed with an average complexity of $\mathcal{O}(m + n)$, where m and n are the number of statements in the compared models. If M and N are sets of statements, the subset A of statements added when going from M to N is simply $A = N - M$, the subset D of deleted statements is $D = M - N$. With a reasonable hash set implementation, each of the containment checks for elements of the subtracted set in the base set have an average complexity of $\mathcal{O}(1)$. This approach to diffing does not attempt to match blank nodes. It favors lower computational complexity and ease of implementation over minimal deltas.

Similarly, applying deltas as a reverse operation is implemented in terms of adding and removing statements from a hash set.

7.4 Push API

Once a user has created a repository, they can start versioning their dataset resources. To create a new revision of a resource, they simply push the RDF description in its current state through straight-forward HTTP requests. An HTTP PUT request introduces or updates a URI-RDF pair, while a DELETE request marks a resource as deleted.

Push access to repositories is authorized via generated API tokens.

Here is a simple example for pushing a resource snapshot from an N-Triples file named `PlanetPluto.nt` via `curl`⁹:

```
curl -H "Authorization: token 46af2e..." \  
      -H "Content-Type: application/n-triples" \  
      -X PUT \  
      --data-binary @PlanetPluto.nt \  
      https://tailr.s16a.org/api/umbel/entities/http://...
```

⁹<http://curl.haxx.se/> (July 12, 2015)

Clients may pass a change timestamp explicitly in their requests. If no timestamp is passed, the current system time is used as a default.

The API accepts various RDF formats as input further reducing the integration effort for dataset maintainers. These formats are parsed, re-serialized and stored in a normalized fashion. The platform stores equivalent RDF models rather than an exact byte-by-byte copy of each RDF description. This allows for detecting malformed input and has several advantages for the internal handling of the data. RDF input is parsed through the Python-bindings for Redland librdf¹⁰.

7.5 User Interface

A browser-based user interface allows users to register and administrate their accounts. Registered users can create dataset repositories and generate Push API tokens in order to start versioning. The platform offers practical advice on how to use the provided interfaces for dataset tracking and revision retrieval (figure 7.4).

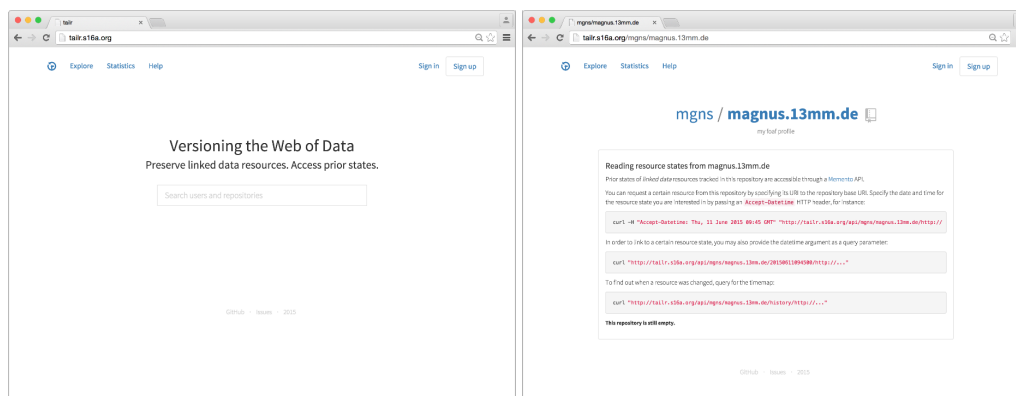


Figure 7.4: The platform's web interface allows users to manage their accounts and repositories, and provides practical hints on how to get started with linked data versioning.

The web interface further provides search capabilities so users can find other users and existing dataset repositories. Each repository page shows a set of

¹⁰<http://librdf.org/> (July 5, 2015)

sample resources. Users can inspect individual resource history visually as a timeline and view prior resource states.

Internally, the user interface for resource history and prior states uses the very same Memento API intended for linked data clients. In a similar manner, additional visualization and inspection tools may be added in the future.

7.6 Integration Points

The system we present aims to solve the problem of keeping linked datasets versioned in an efficient way. However, we have not yet discussed how this system might be integrated with existing linked data tools.

The best results for archiving could obviously be achieved by integrating directly with linked data editors and data sources. Through their application-knowledge, they have a chance of instantly detecting resource changes and pushing updates in an event-based manner (cf. figure 5.1).

We are however aware that this approach is not yet commonly pursued.

There is a lot of existing work on monitoring web resources in general (e.g. [26]) and, more specifically, linked data resources (e.g. [27]). A range of monitoring systems have been developed in an effort to create periodic snapshots of the web of data. Notably the Dynamic Linked Data Observatory (DyLDO) [21] publishes compressed dumps of their weekly linked data crawls. While a very valuable source of historic structured data, the gathered dumps are not very accessible. If instead resource snapshots retrieved by such monitoring approaches were contributed to appropriate dataset repositories on our archival platform, they could be inspected individually through the Memento API.

Even though the platform is only picking up service now, this does not mean that only future revisions of resources can be recorded. Many datasets provide exported snapshots of past versions in some way which can be checked into repositories retrospectively (along with the appropriate date and time). For a start, we plan on creating and pre-filling repositories for well-known datasets so their histories are referenceable and accessible through a consistent, queryable

Implementation

interface.

8 Evaluation

A linked data versioning service would have to serve millions of resource snapshots and handle incoming updates. We have conducted a series of experiments to test how our implementation behaves with regard to Push API response times, Memento API response times and storage growth.

8.1 Test setup

The selected dataset sample and testing infrastructure used in our experiments are described here.

8.1.1 Sample Dataset

For our tests we needed a dataset which was sufficiently large and for which we could retrieve a number of past versions. The *DBpedia*¹ project extracts RDF descriptions on about 4 to 4.5 million entities in its English version alone. It has become one of the central datasets in the Linked Open Data cloud². RDF dumps of past DBpedia releases are available for download.

The test dataset consists of a subset of DBpedia resources in releases 3.2 through 3.9 spanning about four and a half years (i.e. releases 3.2, 3.3, 3.4, 3.5, 3.5.1, 3.6, 3.7, 3.8 and 3.9). A random sample of 1,000,000 resources present in DBpedia releases 3.2 and 3.9 has been selected (see table 8.1).

Descriptions for the selected resources have been extracted from the English DBpedia titles, mapping-based types and properties, collecting statements where

¹<http://dbpedia.org/> (June 27, 2015)

²<http://lod-cloud.net/> (June 27, 2015)

Evaluation

DBpedia Release	Wikipedia Dump Date	#Resources	#Statements	Size tar-gzipped (MiB)
3.2	2008-10-08	1,000,000	4,123,827	99.62
3.3	2009-05-20	974,644	4,272,964	101.10
3.4	2009-09-24	962,113	4,646,814	104.99
3.5	2010-03-16	998,811	4,891,170	106.89
3.5.1	2010-03-16	998,813	4,785,816	108.25
3.6	2010-10-11	998,652	5,338,035	114.35
3.7	2011-07-22	998,503	6,443,269	124.75
3.8	2012-06-01	998,690	6,815,538	127.79
3.9	2013-04-03	1,000,000	5,387,172	123.37
total			46,704,605	1,011.11

Table 8.1: For the experiments, a random subset of 1,000,000 resources present in DBpedia 3.2 and 3.9 was selected. Their descriptions are collected from DBpedia titles, mapping-based types and properties in releases 3.2 through 3.9.

the sampled resources appear in the subject position.

8.1.2 Test Infrastructure

The setup for our evaluation run comprised two machines:

- The first, *tailr*, was a modest 2GB RAM, 2 CPU, 40GB SSD virtual private server (DigitalOcean³ droplet), hosting an instance of our service implementation. The machine ran Ubuntu 14.04 x64 as its operating system. It used the default Python 2.7.6 installation and a standard MariaDB 10.0.19 from the official APT package, with unchanged settings. The versions for the Redland libraries were: libraptor 2.0.15, librasqal 0.9.33, librdf 1.0.17 and language-bindings 1.0.17.1.
- The second machine for pushing and retrieving sample resources to the service repository API was a server hosted at Hasso Plattner Institute and,

³<https://www.digitalocean.com/>

hence, featured a stable and fast internet connection. The rest of its specifications are irrelevant for our measurements.

8.2 Push Performance

In a first study, we measured the time it takes to create new versions of linked data resources.

Method

Going through the release samples in order, the description for each contained resource was pushed to a test repository on the service host. If a resource was not present in an intermediate release of the DBpedia, we issued a `DELETE` request to the API. After the resource snapshots from all releases were submitted, the timing data was parsed from the server logs.

Results

The response times during the test run are summarized in figure 8.1. The exact values are given in table 8.2.

Discussion

Push requests for the first release took the longest time on average. For resources with an empty history, hmap entries need to be created which explains the additional overhead. After that, response times were slightly lower and increased to a small degree with each release.

The growing base+delta chains make for slightly higher costs of reconstructing last-known resource states before diffing. These effects are comparatively small and also limited by chain length restrictions and re-snapshotting for longer histories.

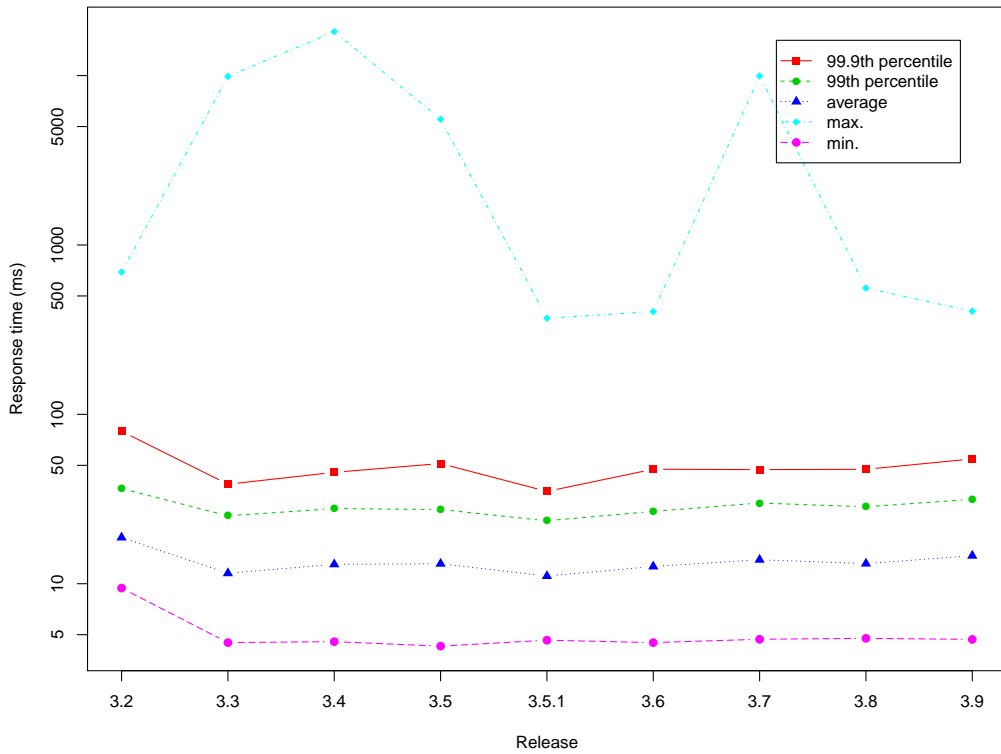


Figure 8.1: Push API response times during the test run, uploading resource snapshots from a sample in 9 releases of the DBpedia.

Because each resource’s history is tracked independently in our storage model, it is further possible to parallelize pushing of resources to reduce the overall time for submitting each release. During the test, average server CPU load was below 15% and disk I/O averaged less than 1.5MB/s leaving plenty of room for parallel requests and greater throughput.

For further push performance improvements, it would be possible to cache the latest resource state and avoid reconstruction costs on resource updates. Diffing could be performed against the cached state instead of reading the base+delta chain.

Evaluation

Rel.	99.9th percentile	99th percentile	Average	Min.	Max.
3.2	79.40	36.48	18.74	9.43	691.70
3.3	38.68	25.17	11.53	4.49	9882.83
3.4	45.45	27.77	13.02	4.55	18160.34
3.5	51.23	27.41	13.12	4.28	5526.19
3.5.1	35.07	23.54	11.10	4.64	369.35
3.6	47.42	26.74	12.64	4.49	404.42
3.7	47.16	29.95	13.85	4.70	9957.81
3.8	47.37	28.46	13.15	4.76	556.37
3.9	54.54	31.50	14.63	4.69	406.42

Table 8.2: Push API response times in milliseconds.

8.3 Storage Analysis

We were further interested in the efficiency of the storage model and the applicability of delta encoding.

Method

After pushing the samples as described previously, the counts for the different cset types up to each release were queried from the database.

Results

The counts for the created snapshot, delta and delete csets after pushing the samples for each of the DBpedia versions are given in table 8.3. The last column lists the cumulated size of the data stored in blob objects.

Discussion

As we can see, the total amount of blob data accounts for about 640 MiB, compared to more than 1,000 MiB for the cumulated size of the individual com-

Evaluation

Rel.	#Snapshots	#Deltas	#Deletes	Aggr. blob data (MiB)
3.2	1,000,000	0	0	157.23
3.3	1,008,781	212,343	25,356	194.75
3.4	1,041,386	492,816	38,938	272.84
3.5	1,128,949	773,436	39,419	360.84
3.5.1	1,129,818	886,766	39,424	378.28
3.6	1,140,421	1,133,508	39,861	421.30
3.7	1,173,592	1,455,516	40,405	502.86
3.8	1,191,561	1,707,166	40,674	550.31
3.9	1,248,696	2,099,056	40,674	637.69

Table 8.3: Number of cset entries by type and aggregate size of the data stored in blobs measured after each release.

pressed dumps in table 8.1. While the overall size of the database is of course larger, it still provides evidence that significant space savings can be achieved by applying delta encoding to dataset revisions, contrasting the growth in the numbers of delta csets and snapshot entries. Though other datasets may possibly change in a different way from our sample, the implemented storage model will nonetheless leverage the benefits of delta compression if they exist.

Also, by storing resources in a database instead of compressed dump files, single resources from a dataset can be updated and queried separately. It is possible to add new versions of a resource without creating a completely new dump. Individual resource snapshots can be retrieved and their timelines traced efficiently. Such interactions are not practicable with monolithic dump files.

8.4 Revision Retrieval

Finally, a linked data versioning platform must be able to efficiently serve resource states. To check how our implementation responds to query load, we performed a third study.

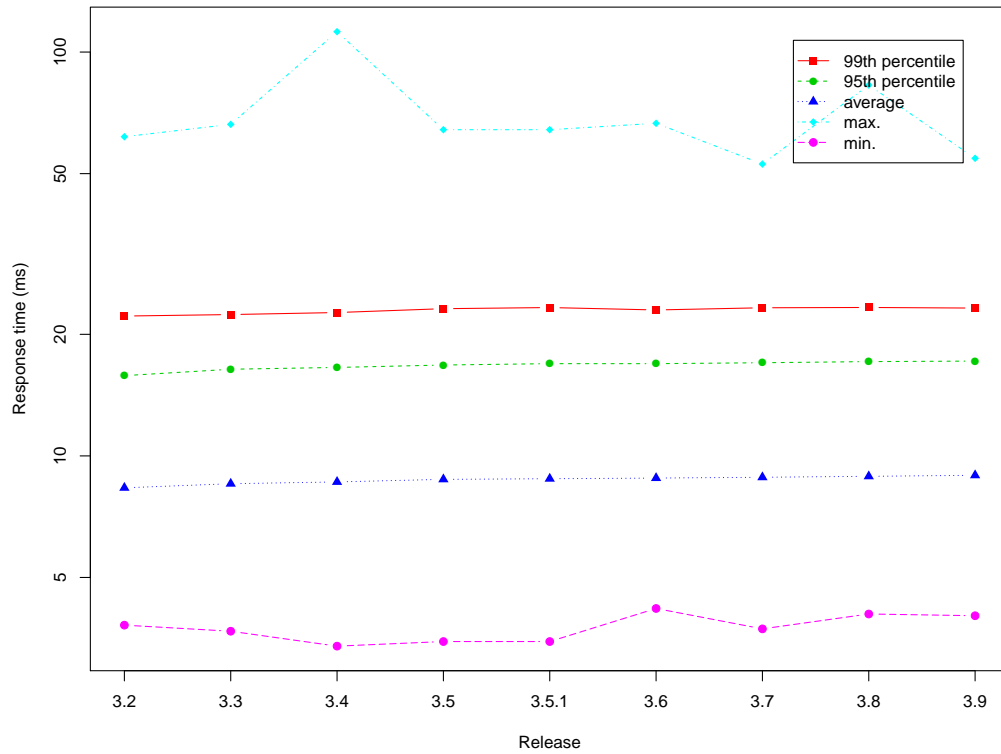


Figure 8.2: Memento API response times (in ms) measured for accessing random revisions of the sample resources.

Method

Memento API performance was measured by requesting each of the sample resources stored in the previous studies in a random revision. The timing data was parsed from the server log files.

Results

The response times grouped by DBpedia release are shown in figure 8.2. The exact values are given in table 8.4.

Evaluation

Rel.	99.9th percentile	99th percentile	Average	Min.	Max.
3.2	22.20	15.80	8.34	3.81	61.71
3.3	22.39	16.39	8.53	3.68	66.23
3.4	22.65	16.56	8.62	3.38	112.31
3.5	23.15	16.79	8.75	3.47	64.23
3.5.1	23.30	16.92	8.78	3.47	64.23
3.6	22.98	16.92	8.81	4.19	66.62
3.7	23.27	17.01	8.85	3.73	52.81
3.8	23.31	17.12	8.90	4.06	82.86
3.9	23.23	17.17	8.95	4.02	54.57

Table 8.4: Memento API response times in milliseconds.

Discussion

We can see a slight increase in the response times for later revisions. Again, this is due to the longer base+delta chains resulting in higher resource reconstruction costs. Response times are still in a very acceptable range and our re-snapshotting strategy will limit retrieval costs for longer dataset histories.

8.5 Remarks

During our initial test runs with an earlier version of the prototype system, push performance was largely limited by RDF parsing. About 82% of the processing time was spent parsing. After switching from the pure-Python *rdflib*⁴ to language-bindings for the C-based *librdf*⁵, parsing time was reduced to about 26% on the same dataset.

With regard to delta compression, for datasets containing blank nodes additional space savings could be achieved by performing blank node matching (cf. [28]). By finding an isomorphism between blank nodes of two resource revisions, the system could potentially store smaller deltas at the cost of computa-

⁴<https://github.com/RDFLib/rdflib> (July 3, 2015)

⁵<http://librdf.org/> (July 3, 2015)

tional overhead. Many RDF tools and stores, however, already assign identifiers to blank nodes that – given an RDF syntax like N-Triples – are transmitted to and used by our current hash-based diff implementation.

9 Future Work

This chapter outlines possible directions for future work on the versioning platform itself and further usage of the gathered data.

9.1 Platform Enhancements

The created platform may be enhanced in different ways.

For additional performance gains, the platform could employ caches on different levels. Prior resource states never change and thus memento responses for frequently requested resources could be effectively cached. Similarly, caching of the most recent states for tracked resources would allow for faster Push API responses, as it avoids reading base+delta chains for diffing and further processing.

To make pushing of complete dataset states more efficient, the platform could offer a bulk import mechanism. A streaming push API could allow use-cases where single resources have unusually high update frequencies, e.g. sensor data.

At some point, distributing the platform implementation may be of interest to accommodate for increasing loads. At the moment though that is not necessary.

9.2 Evaluation of Alternative Backends

Because none of the existing solutions we have investigated fit our intent of building a linked data versioning platform, our approach adapted ideas from

version control systems for our storage backend. As mentioned in chapter 6, there are other candidate concepts which could be analyzed.

One such alternative would be to instead implement an extended quad table schema which supports our repository concept and performs tuple versioning. This approach likely has different characteristics that would need to be compared with our current solution. The storage model we are using now is geared towards simple linked data resource queries with evaluation happening on the client side. A modified quad table schema could potentially answer more complex queries if implemented efficiently. Care must be taken though to avoid problems similar to those common with triplestores and SPARQL endpoints.

Another class of storage systems that could possibly support a versioning platform for linked data are graph databases with versioning capabilities, Neo4J¹ for instance. It would be interesting to see, whether and how the versioning concepts and query constructs of such graph databases could be mapped appropriately to our approach. If so, what are the trade-offs?

Independent of the used storage backend, the Memento API would provide the very same interface to prior resource states. The change would be transparent for clients.

9.3 Monitoring Service

Our current platform is primarily concerned with storing linked data resource revisions and efficient, granular access. Though it is straight-forward to push new resource snapshots, dataset maintainers still need to take care of this. A complementary monitoring service could make versioning of linked data resources even more convenient.

Such a service would allow developers to set a target dataset for monitoring, e.g. by providing it with the domain/base URL of the dataset or by uploading a semantic sitemap [29]. The monitoring service would then periodically check targeted resources for changes and push updated resource states to the version-

¹<http://www.neo4j.org/graphgist?608bf0701e3306a23e77> (July 7, 2015)

ing service.

9.4 Analysis Tools

The historic information accumulated and exposed with the help of the service provides great potential for new linked data applications.

The aggregated knowledge about data resources and changes could be used to analyze quality of linked datasets for instance. The resulting quality metrics could be used by data engineers to identify problems in the data and improve it. The influence of their changes can then be verified by comparing against prior dataset versions in the system.

Other tools could visualize dynamic processes captured as linked data through time and correlate that information. Such visualizations may create a better understanding of the underlying processes, explain how and why certain indicators evolved over time.

10 Summary

This work introduced an approach to versioning datasets based on linked data principles. It specifically addresses discoverability and accessibility issues prevalent in today's versioning practice on the semantic web.

The approach further aims for a separation of concerns regarding linked data publishing and archival of past resource states. This distinction enables a service-oriented approach towards linked data versioning. Linked data publishers can more easily start versioning their datasets in a way which allows granular access to individual resource histories, making retrospective investigation feasible.

We presented a design for a linked data archival platform following this approach with interfaces for submitting snapshots of linked data resources as well as a time-based access protocol for resource revisions. The openness of the platform and straight-forward APIs enable it to be integrated with existing linked data publishing infrastructures. Explicit references from original resources to their prior states allow clients to follow such links and inspect the resource history.

We have implemented the presented design as a Python web application which can be used to provide linked data archival as a service to publishers of linked data. The platform encodes resource revisions as a series of deltas based on periodic snapshots. This model balances the requirements for storage space and retrieval time.

Our evaluation showed, that the implementation can support large datasets, efficiently handling the creation of new resource revisions and retrieval of past resource states.

In the future, the service could form the basis for analysis and visualization

Summary

tools, providing additional insights into the data by putting it into its historic context and clarifying dynamic developments. Complementary services, based on existing work, could offer monitoring of linked datasets to developers, using our platform as a storage backend.

Part of this work has been submitted to the 11th International Conference on Semantic Systems (SEMANTiCS), Research Track 2015.

Bibliography

- [1] A. Zaveri, A. Rula, A. Maurino, R. Pietrobon, J. Lehmann, S. Auer, and P. Hitzler, "Quality Assessment Methodologies for Linked Open Data," *Semantic Web Journal*, 2014.
- [2] C. B. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche, "SPARQL Web-Querying Infrastructure: Ready for Action?," in *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference*, pp. 277–293, 2013.
- [3] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, vol. 284, pp. 34–43, May 2001.
- [4] T. Berners-Lee, "Linked Data." W3C Design Issues, <http://www.w3.org/DesignIssues/LinkedData.html>, July 2006.
Accessed June 22, 2015.
- [5] T. Heath and C. Bizer, *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 1st ed., 2011.
- [6] F. Michel, J. Montagnat, and C. Faron-Zucker, "A survey of RDB to RDF translation approaches and tools," research report, I3S, May 2014.
ISRN I3S/RR 2013-04-FR 24 pages.
- [7] M. Knuth, J. Hercher, and H. Sack, "Collaboratively Patching Linked Data," in *Proc. of 2nd Int. Workshop on Usage Analysis and the Web of Data (USEWOD 2012), co-located with the 21st International World Wide Web Conference 2012 (WWW 2012), Lyon (France)*, April 2012.
- [8] T. Berners-Lee and D. Connolly, "Delta: an ontology for the distribution of differences between RDF graphs." W3C Design Issues, <http://www.w3.org/DesignIssues/lncs04/Diff.pdf>, March 2004.
Accessed June 22, 2015.
- [9] C. Gutierrez, C. A. Hurtado, and A. Vaisman, "Introducing Time into RDF,"

Bibliography

- IEEE Transactions on Knowledge and Data Engineering*, vol. 19, pp. 207–218, Feb. 2007.
- [10] S. Auer and H. Herre, “A Versioning and Evolution Framework for RDF Knowledge Bases,” in *Proceedings of the 6th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*, vol. 4378 of *PSI’06*, (Berlin, Heidelberg), pp. 55–69, Springer-Verlag, June 2006.
- [11] H. V. de Sompel, M. L. Nelson, R. Sanderson, L. Balakireva, S. Ainsworth, and H. Shankar, “Memento: Time Travel for the Web,” *CoRR*, vol. abs/0911.1112, 2009.
- [12] H. V. de Sompel, R. Sanderson, M. L. Nelson, L. Balakireva, H. Shankar, and S. Ainsworth, “An HTTP-Based Versioning Mechanism for Linked Data,” in *Proceedings of Linked Data on the Web (LDOW2010)*, (Raleigh, USA), April 2010.
- [13] S. Cassidy and J. Ballantine, “Version Control for RDF Triple Stores,” in *ICSOFT 2007, Proceedings of the Second International Conference on Software and Data Technologies, Volume ISDM/EHST/DC, Barcelona, Spain*, pp. 5–12, July 2007.
- [14] M. V. Sande, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens, and R. V. de Walle, “R&Wbase: git for triples,” in *LDOW (C. Bizer, T. Heath, T. Berners-Lee, M. Hausenblas, and S. Auer, eds.)*, vol. 996 of *CEUR Workshop Proceedings*, ceur-ws.org, 2013.
- [15] M. Graube, S. Hensel, and L. Urbas, “R43ples: Revisions for Triples - An Approach for Version Control in the Semantic Web,” in *Proceedings of the 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems, LDQ@SEMANTiCS*, 2014.
- [16] S. M. Embury, B. Jin, S. Sampaio, and I. Eleftheriou, “On the Feasibility of Crawling Linked Data Sets for Reusable Defect Corrections,” in *Proceedings of the 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems, LDQ@SEMANTiCS*, September 2014.
- [17] B. Thomas, V. Pierre-Yves, and V. Bernard, “Requirements for vocabulary preservation and governance,” *Library Hi Tech*, vol. 31, no. 4, pp. 657–668, 2013.
- [18] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, and G. Tum-

- marello, "Sindice.com: a Document-Oriented Lookup Index for Open Linked Data," *Int. J. Metadata Semant. Ontologies*, vol. 3, pp. 37–52, November 2008.
- [19] R. Isele, J. Umbrich, C. Bizer, and A. Harth, "LDspider: An Open-source Crawling Framework for the Web of Linked Data," in *Proceedings of the ISWC 2010 Posters & Demonstrations Track: Collected Abstracts, Shanghai, China*, November 2010.
- [20] A. Hogan, A. Harth, J. Umbrich, S. Kinsella, A. Polleres, and S. Decker, "Searching and Browsing Linked Data with SWSE: The Semantic Web Search Engine," *Web Semant.*, vol. 9, pp. 365–401, December 2011.
- [21] T. Käfer, J. Umbrich, A. Hogan, and A. Polleres, "Towards a Dynamic Linked Data Observatory," in *LDOW 2012 - Linked Data on the Web* (C. Bizer, T. Heath, T. Berners-Lee, and M. Hausenblas, eds.), (Lyon, France), April 2012.
- [22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
- [23] S. Chacon, *Pro Git*. Berkely, CA, USA: Apress, 1st ed., 2009.
- [24] B. O'Sullivan, *Mercurial: The Definitive Guide*. O'Reilly Media, June 2009.
- [25] K. Stefanidis, I. Chrysakis, and G. Flouris, "On Designing Archiving Policies for Evolving RDF Datasets on the Web," in *Conceptual Modeling - 33rd International Conference, ER 2014, Atlanta, GA, USA*, pp. 43–56, 2014.
- [26] "Common Crawl." Online, Mar. 2012.
- [27] A. Harth, "Billion Triples Challenge data set." Downloaded from <http://km.aifb.kit.edu/projects/btc-2012/>, 2012.
- [28] Y. Tzitzikas, C. Lantzaki, and D. Zeginis, "Blank Node Matching and RDF/S Comparison Functions," in *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, pp. 591–607, 2012.
- [29] R. Cyganiak, H. Stenzhorn, R. Delbru, S. Decker, and G. Tummarello, "Semantic Sitemaps: Efficient and Flexible Access to Datasets on the Semantic Web," in *The Semantic Web: Research and Applications, 5th Euro-*

Bibliography

pean Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings, pp. 690–704, June 2008.

I hereby certify that the material contained in this thesis is my own work and does not contain significant portions of unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation.

Hiermit versichere ich, dass diese Arbeit selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, July 13, 2015

(Paul Meinhardt)