TailR: A Platform for Preserving History on the Web of Data

Paul Meinhardt
Hasso Plattner Institute,
University of Potsdam
Prof.-Dr.-Helmert-Str. 2–3
14482 Potsdam, Germany
paul.meinhardt
@student.hpi.de

Magnus Knuth
Hasso Plattner Institute,
University of Potsdam
Prof.-Dr.-Helmert-Str. 2–3
14482 Potsdam, Germany
magnus.knuth@hpi.de

Harald Sack
Hasso Plattner Institute,
University of Potsdam
Prof.-Dr.-Helmert-Str. 2–3
14482 Potsdam, Germany
harald.sack@hpi.de

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 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 how changes propagate through 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 dataset 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.

In this paper, we describe a basic model view for linked datasets and a platform for preserving the history of arbitrary linked datasets over time, providing access to prior states of contained resources via mementoes.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Linked Data, RDF, Versioning, History, Memento

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 documents", 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, the 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 particu-

SEMANTiCS '15, September 15 - 17, 2015, Vienna, Austria

ISBN 978-1-4503-3462-4/15/09…$15.00

DOI: http://dx.doi.org/10.1145/2814864.2814875

lar, temporal attributes like recency, volatility, and timeliness [14] 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 can we later identify patterns in changes and understand how they propagate, see which new resources are referenced and how they grow [12].

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 problems [1]. When data publishers go the extra mile for archiving past dataset states, different solutions expose different, non-standard interfaces for accessing these, 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.

The platform we propose aims at separating the concerns of data publishing, monitoring, and recording linked dataset history. This separation frees dataset publishers from the burden of maintaining a versioning solution for their datasets themselves and introduces a common version interface to record datasets. Ideally, data publishers will submit changes to an official repository for their own dataset. For datasets where this is not the case, others can push change information to the platform from existing linked data crawling and monitoring tools. Through the creation of a community-maintained storage platform for historic dataset states, in some regards similar to the Internet Archive[1], 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 the web of data. To that end, the proposed platform establishes a consistent time-travel interface for all versioned datasets that aligns well with linked data principles.

[1]https://archive.org/web/

The remainder of this paper is structured as follows: Section 2 discusses related work in the area of versioning semantic data and version access strategies. Section 3 further motivates and describes our approach to archiving, the backing storage model and service implementation. We share insights gained in first experiments with the platform in section 4. Finally, we conclude the paper with closing remarks on the presented work and an outlook onto future directions.

2. RELATED WORK

A time travel protocol for the web has been previously proposed by the Memento Project[2] [4], 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 Archive, and public libraries. It is available as an extension to MediaWiki instances[3] and a proxy implementation exists for 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 [5]. To achieve this, DBpedia releases were stored in a MySQL database as independent snapshots and served through a *timegate* endpoint (a more detailed description of the protocol follows in section 3.5). The demonstrator however misses an archiving interface and updates have been discontinued with DBpedia 3.9. While our platform analogously adopts the Memento framework for version access, it aims to become an archival platform not just for the DBpedia, but other datasets as well. As such, it openly provides dataset archival as a service and comprises a Push API that allows users to submit new dataset revisions. The backing storage concepts are more efficient so it will be able to keep up with many datasets and a growing number of revisions.

Revision control systems such as CVS, Subversion, Mercurial, and Git are well-established tools in software engineering for versioning mainly plain text files, such as source code or CSV data[4]. Such versioning systems are typically based on a unique serialization format. Using an appropriate RDF serialization (preferably a fixed-order line-based format, e.g. sorted N-Triples) such systems can equally well be used for versioning RDF data [3]. This approach is practicable for small datasets but has its limitations due to file-orientation and revision control system characteristics. Additionally, an extra HTTP layer is needed on top of repositories in order to expose the recorded revision information and make it dereferenceable.

A variety of formats and vocabularies allow for describing updates to RDF datasets: Both *Delta* [3] and *RDF Patch*[5] propose dedicated patch file formats. *Changeset*[6] defines a vocabulary for changes to resource descriptions using RDF reification including terms for metadata on the change. Finally, the *Graph Update Ontology (GUO)*[7] provides an OWL vocabulary for graph changes which avoids RDF reification. These formats are primarily made for change propagation between multiple copies of the same dataset. Except for

[2]http://mementoweb.org
[3]http://www.mediawiki.org/wiki/Extension:Memento
[4]http://blog.okfn.org/2013/07/02/
git-and-github-for-data/
[5]http://afs.github.io/rdf-patch/
[6]http://vocab.org/changeset/schema.html
[7]http://purl.org/hpi/guo#

RDF Patch they produce rather bloated and verbose descriptions of change information. Based on such change descriptions historic versions of RDF datasets could be reconstructed. But, none of these vocabularies developed to an accepted standard, and neither is the publication of dataset updates commonly adopted.

SemVersion [13] provided versioning for RDF models and ontologies based on structural and semantical deltas. Another versioning approach for RDF was introduced by [2] and integrated into the collaborative RDF editor Powl. Both systems target ontology evolution and are meanwhile discontinued.

R&Wbase [10] tracks changes within a modified triplestore. It encodes deltas via context values assigned to the changed triples (or quads) for each revision: Additions are marked by even context values, deletions receive odd values. In order to restore a revision, triples are scanned and selected if the highest context value found is even. R&Wbase allows querying via SPARQL and virtual graphs for revisions.

R43ples [6] implements an RDF-versioning proxy. It uses named RDF graphs to group additions and deletions. Revisions of the data are restored by creating a temporary graph from the head revision, then rolling back changes stored in the addition/deletion graphs. All 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. R43ples also introduces non-standard SPARQL keywords for querying past states of a dataset.

Both R&Wbase and R43ples do not consider the linked data aspect of versioned datasets, i.e. how to dereference resource URIs and retrieve an RDF description for a specific point in time. Linked data interfaces to SPARQL endpoints, like the popular Pubby[8], are not version-aware. They do not allow for time queries and would have to be adapted for use with R&Wbase or R43ples stores. They also rely on a systematic, defined mapping to query statements related to a URI. This mapping would need to be versioned as well or guaranteed to never change. Otherwise a set of statements retrieved from a URI might not be equivalent to what would have been retrieved earlier. Finally, there are linked datasets which do not originate from data sources exposed through a SPARQL endpoint. For such datasets these approaches would not work.

Recording changing states of web resources is a task the Internet Archive and several other web archives have committed themselves to for a long time. They monitor and preserve various artifacts on the web and provide access to prior versions. The Dynamic Linked Data Observatory [9, 8] monitors a subset of the web of data on a weekly basis, and provides data dumps and statistics on a two-hop neighborhood of eighty thousand linked data documents.

These initiatives make it possible to gain insights on the dynamics of web data but only show selected fragments of the data in coarse time intervals. Both the monitored documents and time intervals can not be managed by the data consumers.

3. APPROACH

Existing work on versioning semantic data models focuses primarily on tracking changes to RDF datasets. While a similar sounding concept, the notion of an RDF dataset is not the same as that of a *linked dataset*. The purpose of an RDF dataset is solely knowledge representation, it is not concerned with how modeled information is published or whether used resources can be dereferenced at all. In a linked dataset, RDF descriptions are bound to resolvable URIs, allowing agents to retrieve resource descriptions using HTTP.

[8]http://wifo5-03.informatik.uni-mannheim.de/pubby/

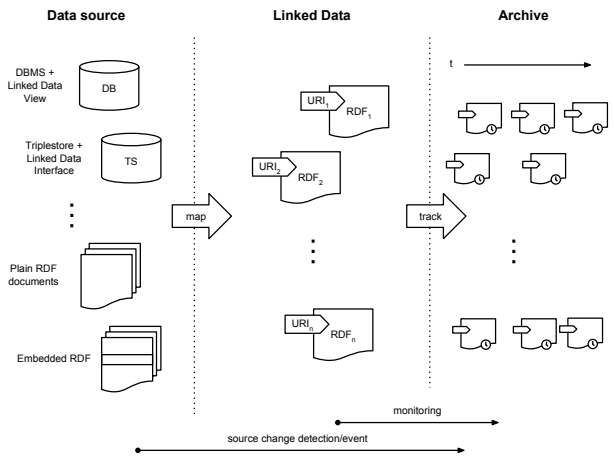

Figure 1: Linked datasets originate from a variety of different data sources. A versioning system receives notifications about changes to a dataset directly from the publisher or through a monitoring tool.

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 1).

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

3.1 A Model for Linked Datasets

A linked dataset is a set of URIs and associated RDF descriptions (returned when dereferencing a URI):

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

Above definition provides a common view onto linked datasets independent of the underlying data source. It is suitable for implementing a versioning scheme which applies to all kinds of linked datasets.

3.2 Possible Changes in Linked Datasets

For tracking the history of a linked dataset, we can identify these possible (atomic) changes:

- introduction of a URI-RDF description pair
- 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.

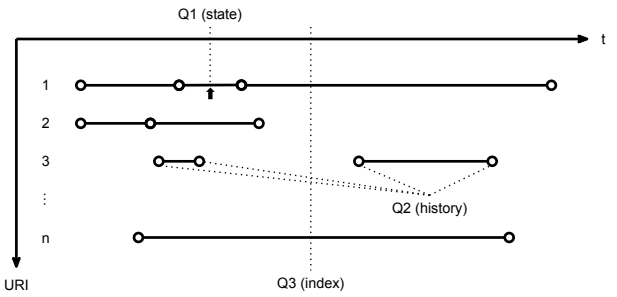

Figure 2: Linked dataset history consisting of individual timelines for each URI and visualization of basic queries.

3.3 Querying Linked Dataset History

A linked data archival system should answer the following essential queries (in order of importance):

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 2. Q1 allows the retrieval of a past state of a specific resource, a so-called "memento". We consider this to be the most common type of query. Q2 allows for tracing changes to a single resource through time, figuring out at what points it was actually changed. Q3 asks for an index of the resources represented in a dataset at a certain point in time.

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

3.4 Service Overview

The platform 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 and a read-only Memento API for accessing stored version information of linked data resources.

3.5 Memento API

The idea of a time-travel interface for the web has previously been developed and advocated by Van de Sompel et al. [4]. Their protocol-based framework, called "Memento", uses standard HTTP capabilities in order to link and retrieve past states of web resources[9]. The archival platform we propose applies the concepts of Memento to linked data resources for consistent and discoverable version access.

Each repository acts as a *timegate* for the original resources. It supports datetime negotiation for accessing prior resource states, *mementoes*. This 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).The server will reply with an RDF description equivalent to that of the resource valid at the given point in time. It will also include a `Memento-Datetime` response header informing the client about when the resource was actually changed (might be earlier than the 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 query parameter instead of a header (`?datetime=...`).

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.

If a client is interested in the change history of a particular resource (Q2), 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 Memento framework makes use of the HTTP `Link` header[10] for supplying URI references between the different kinds of endpoints. The service memento responses will always contain the URIs of the corresponding original resource and timemap. For best integration, the original linked data endpoint should refer to the repository as its timegate by inserting an appropriate `Link` header (`rel="timegate"`). A simple Apache or Nginx proxy configuration change will often suffice to achieve this explicit reference.

A major advantage we see 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 and is easily integrated with existing linked datasets.

3.6 Storage Model

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

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 based model is 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, snapshots and deltas, to balance space requirements and retrieval time (cf. [11]).

The storage structure is designed to support memento queries. Basic entities in the system are (see also figure 3):

repos Repositories are created by users and are typically referenced by name. Conceptually, each repository encapsulates a linked dataset and its history.

[9]http://tools.ietf.org/html/rfc7089

[10]https://tools.ietf.org/html/rfc5988

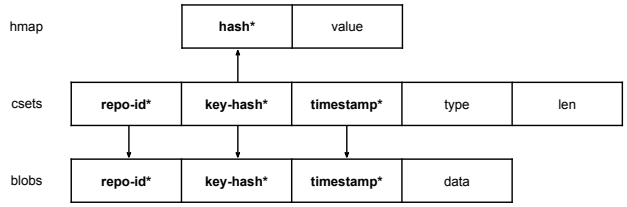

Figure 3: Storage schema used to record dataset changes. A $*$ indicates a primary key component.

csets Changesets encode information about modifications to linked data resources within a repository at a particular point in time. There are three types of changeset entries: *snapshot*, *delta* and *delete*. Each 32-byte cset entry is identified and indexed by repository id, a hash of the original resource URI (the "storage key") and its timestamp.

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. The URI hashes stored in the hmap are typically much shorter than storing the plain URI as part of each cset redundantly. Also, queries 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.

Notice that csets and blobs are actually de-normalized. They share the same primary key but are handled as different entities. Queries Q2 and Q3 can already be answered without reading any blob data. We thus avoid scanning more data than necessary by storing blobs separately. While our current implementation uses the same database system for storing all of the above entities, this structure also opens up the possibility to move blobs to a dedicated blob store should that prove beneficial.

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. This is efficient because of the chosen index/primary key. Queried cset entries for resource reconstruction may be read sequentially from a coherent range.

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.

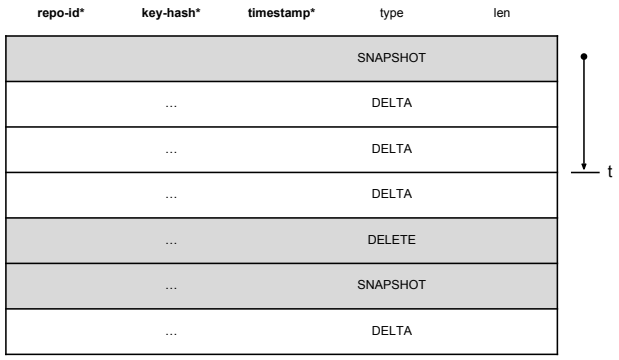

Figure 4: Base+delta chain: Any resource state is reconstructed from the latest non-delta cset prior to the requested t and subsequent deltas.

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 URIs may be very long for linked datasets making it infeasible to store and version it directly.

3.7 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 (snapshots are always available, while deltas are usually not). An HTTP `PUT` request introduces or updates a URI-RDF pair, while a `DELETE` request marks a resource as deleted. 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 (including RDF/XML, N-Triples, Turtle, N-Quads, RDFa, RDF/JSON, TriG[11]) as input further reducing the integration effort for dataset maintainers. Push access to repositories is authorized via generated API tokens.

3.8 Implementation

The described protocols and storage model have been implemented as a Python web service on top of a data store.

The hmap, csets and blob entities are stored by a relational database management system (MariaDB) with the option of moving out blobs to a dedicated blob store should that prove more fitting and the need arises. To further scale the service we consider sharding the database once demands outgrow our current setup and the gains we can get by further fine-tuning. Depending on the actual workloads we will see, sharding by the combination of repository and key-hash might be appropriate, if we see a focus on Q1 and Q2 queries, or by repo only, if analysis deems Q3 to be highly relevant.

As for hashing of storage keys, we are using the SHA-1 hash function also used in popular DVCS systems like Git or Mercurial. 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.

Like stated before, the push API accepts a number of RDF input formats. 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 librdf[12].

Snapshot blobs contain RDF data as zlib-compressed N-Quads[13], data for deltas is stored in zlib-compressed RDF-Patch format[14]. Both of these formats are syntactically very similar, making it straight-forward to generate patches based on two N-Quad representations and, reversely, reconstructing models from an N-Quad snapshot and RDF-Patch deltas. Zlib-compression effectively reduces the required storage capacity and transfer costs for blob data with moderate computational overhead.

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.

The source code is published as a GitHub repository[15], an instance of the service is available at http://tailr.s16a.org/.

[11]for supported MIME types see http://librdf.org/raptor/api/raptor-formats-types-by-parser.html
[12]http://librdf.org/
[13]http://www.w3.org/TR/n-quads/
[14]http://afs.github.io/rdf-patch/
[15]https://github.com/pmeinhardt/tlr

Rel.	Wikipedia Dump Date	#Resources	#Statements	Dump size (KiB)
3.2	2008-10-08	100,000	416,303	10,904
3.3	2009-05-20	97,461	431,895	11,062
3.4	2009-09-24	96,180	469,529	11,493
3.5	2010-03-16	99,876	481,245	11,930
3.5.1	2010-03-16	99,876	491,987	12,103
3.6	2010-10-11	99,838	537,401	12,771
3.7	2011-07-22	99,842	648,320	13,950
3.8	2012-06-01	99,867	684,965	14,253
3.9	2013-04-03	100,000	540,237	13,301

Table 1: For evaluation a random subset of 100,000 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.

3.9 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. 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 and, more specifically, linked data resources. 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) [9] 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 interface.

4. EXPERIMENTAL RESULTS

A direct comparison of the system with other related approaches is currently not possible, mostly because they do not provide linked data interfaces with time-travel capabilities. This section contains experimental results for our performance measurements of the Push and Memento APIs as well as the storage model.

The evaluation dataset consists of a subset of DBpedia resources in releases 3.2 through 3.9 (3.2, 3.3, 3.4, 3.5, 3.5.1, 3.6, 3.7, 3.8 and 3.9). A random sample of 100,000 resources

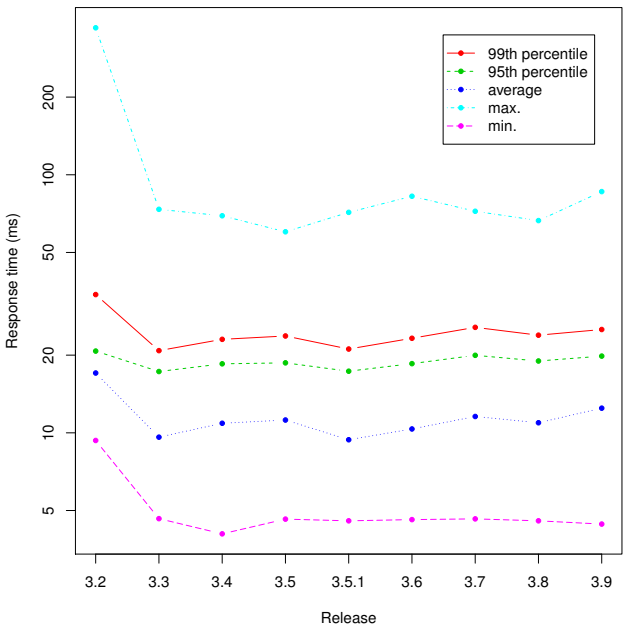

Figure 5: Push API response times (in ms) during the evaluation run with 100,000 resources across 9 versions of DBpedia.

present in DBpedia releases 3.2 and 3.9 has been selected (see table 1). Descriptions for the selected resources have been extracted from the English DBpedia titles, mapping-based types and properties, collecting statements where the sampled resources appear in the subject position. The sample dataset used in our evaluation is publicly available for future reference[16].

The setup for our evaluation run comprised two virtual private servers (DigitalOcean[17] droplets) in the same datacenter. The first, *tailr*, was a 2GB RAM, 2 CPU, 40GB SSD machine, hosting an instance of our service implementation. The second, *pushr*, was a 1GB RAM, 1 CPU, 30GB SSD machine for pushing sample resources to the service repository API. Both machines were running Ubuntu 14.04 x64 with the default Python 2.7.6.

In a first study, we measured Push API response times. Going through the release samples in order, the description for each contained resource was pushed to a test repository. If a resource was not present in a version of the DBpedia, we issued a `DELETE` request to the API. The response times during the test run are summarized in figure 5. The timing data was parsed from the server logs.

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 increased to a small degree. 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.

Server CPU load during the whole experiment was below 20%, disk I/O averaged less than 2MB/s (never exceeded 3MB/s) leaving plenty of room for parallel requests and greater throughput.

[16]http://s16a.org/files/tailr/testdata.tgz
[17]https://www.digitalocean.com/

Rel.	#Snapshots	#Deltas	#Deletes	Data size (KiB)
3.2	100,000	0	0	16182.14
3.3	100,878	21,350	2,539	20047.63
3.4	104,146	49,716	3,920	28131.71
3.5	112,863	78,134	3,967	37200.18
3.5.1	112,964	89,423	3,967	38986.99
3.6	114,006	114,380	4,030	43423.59
3.7	117,337	146,922	4,075	51857.38
3.8	119,140	172,405	4,099	56732.34
3.9	124,933	211,744	4,099	65730.81

Table 2: Number of cset entries by type and aggregate size of the data stored in blobs measured after pushing each release.

We were further interested in the efficiency of the storage model and the applicability of delta encoding. The counts for the created snapshot, delta and delete csets after pushing the samples for each of the DBpedia versions are given in table 2. The last column lists the cumulated size of the data stored in blob objects.

As we can see, the total amount of blob data accounts for 64.19 MiB, compared to 109.15 MiB for the cumulated size of the individual compressed dumps in table 1 (or 579.70 MiB for the uncompressed N-Triples files). Contrasting the growth in the numbers of delta csets with that of snapshot entries, we can gather that seen space savings can be largely attributed to delta encoding. 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.

Finally, we measured the Memento API performance by requesting each of the sample resources in a random revision. Response times grouped by DBpedia release are shown in figure 6. The data was again parsed from the server log files.

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.

5. CONCLUSIONS AND FUTURE WORK

In this paper we introduced a model for versioning linked datasets and presented *TailR*, an archiving system for linked data resources. An implementation of the presented concepts and service is openly available. We are confident that providing "linked data archival as a service" will lower the entry barrier for dataset maintainers to start recording the history of their datasets. Consequently, we plan to accumulate a significant amount of historic information on public datasets and facilitate research by providing this information through standard protocols and interfaces.

The platform will be the basis for new, time-based linked data applications. Future work includes added services based on the recently recognized and aggregated changes, an inte-

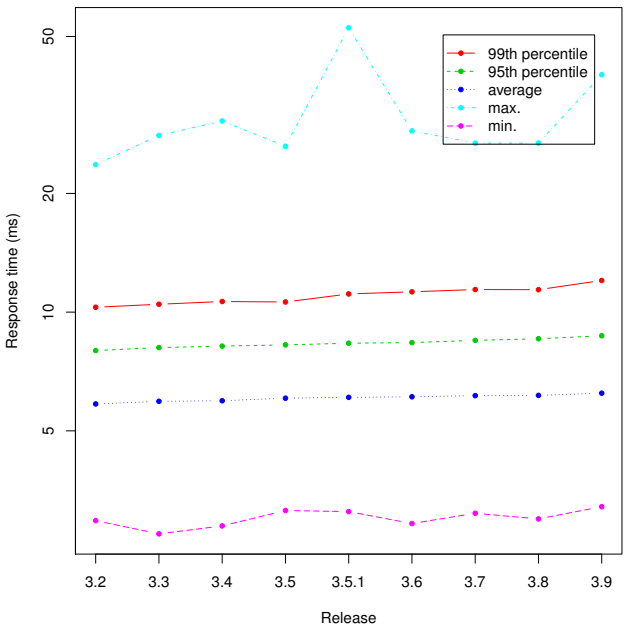

Figure 6: Memento API response times (in ms) measured for accessing random revisions of the sample resources.

grated notification mechanism (e.g. via PubSubHubbub[18]) and resource statistics. Since our approach does not support querying of multiple resources per se, it needs to be researched which query approaches can be applied for within this setting. Since the whole dataset is not accessible at once, traversal based querying [7] seems a good candidate.

6. ACKNOWLEDGMENTS

This work was partially funded by the German Government, Federal Ministry of Education and Research for the project D-Werft (03WKCJ4D).

7. REFERENCES

[1] 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*, pages 277–293, 2013.

[2] 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*, volume 4378 of *PSI'06*, pages 55–69, Berlin, Heidelberg, June 2006. Springer-Verlag.

[3] T. Berners-Lee and D. Connolly. Delta: an ontology for the distribution of differences between RDF graphs, 2004.

[4] H. V. de Sompel, M. L. Nelson, R. Sanderson, L. Balakireva, S. Ainsworth, and H. Shankar. Memento: Time travel for the web. *CoRR*, abs/0911.1112, 2009.

[18]https://github.com/pubsubhubbub/PubSubHubbub

[5] 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.

[6] 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, LDQ2014*, volume 1215 of *CEUR Workshop Proceedings*, Leipzig, Germany, September 2014. CEUR-WS.org.

[7] O. Hartig. SPARQL for a web of linked data: Semantics and computability. In *The Semantic Web: Research and Applications - 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings*, pages 8–23, 2012.

[8] T. Käfer, A. Abdelrahman, J. Umbrich, P. O´Byrne, and A. Hogan. Observing linked data dynamics. In *The Semantic Web: Semantics and Big Data*, pages 213–227. Springer, 2013.

[9] T. Käfer, J. Umbrich, A. Hogan, and A. Polleres. Towards a dynamic linked data observatory. In C. Bizer, T. Heath, T. Berners-Lee, and M. Hausenblas, editors, *LDOW 2012 - Linked Data on the Web*, Lyon, France, April 2012.

[10] M. V. Sande, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens, and R. V. de Walle. R&Wbase: Git for triples. In C. Bizer, T. Heath, T. Berners-Lee, M. Hausenblas, and S. Auer, editors, *LDOW*, volume 996 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[11] 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*, pages 43–56, 2014.

[12] J. Umbrich, M. Hausenblas, A. Hogan, A. Polleres, and S. Decker. Towards dataset dynamics: Change frequency of linked open data sources. In *LDOW2010*, Raleigh, USA, April 2010.

[13] M. Völkel and T. Groza. Semversion: An rdf-based ontology versioning system. In *Proceedings of IADIS International Conference on WWW/Internet*, volume 1, pages 195–202, Murcia, Spain, Oktober 2006. IADIS, IADIS.

[14] A. Zaveri, A. Rula, A. Maurino, R. Pietrobon, J. Lehmann, and S. Auer. Quality assessment for linked data: A survey. *Semantic Web Journal*, 2015.