Exposing RDF Archives using Triple Pattern Fragments

Ruben Taelman, Ruben Verborgh, and Erik Mannens

imec - Ghent University - IDLab
Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium
{firstname.lastname}@ugent.be

Abstract. Linked Datasets typically change over time, and knowledge of this historical information can be useful. This makes the storage and querying of Dynamic Linked Open Data an important area of research. With the current versioning solutions, publishing Dynamic Linked Open Data at Web-Scale is possible, but too expensive. We investigate the possibility of using the low-cost Triple Pattern Fragments (TPF) interface to publish versioned Linked Open Data. In this paper, we discuss requirements for supporting versioning in the TPF framework, on the level of the interface, storage and client, and investigate which trade-offs exist. These requirements lay the foundations for further research in the area of low-cost, Web-Scale dynamic Linked Open Data publication and querying.

Keywords: Linked Data, versioning, Triple Pattern Fragments, Linked Data Fragments, SPARQL

1 Introduction

The ability to perform both simple and complex queries over Linked Datasets is of utmost importance for gaining insights into data. Data analysis can be time-agnostic, but historical and real-time analysis requires probing data *at* certain points in time, or *over* time periods. For example, retrieving biomedical patient information at a certain point in time, or analyzing the evolution of a disease over time.

RDF [1] and SPARQL [5] allow us to represent and query Linked Data. Most Linked Datasets are dynamic on dataset, schema and/or instance level [6]. These refer to additions, changes or deletions of respectively complete datasets, ontologies and separate facts.

A survey on archiving Linked Open Data [3] motivates the relevance and importance of this domain. The authors point out that the current approaches have scalability drawbacks because they are not sufficiently designed or applied to the required Web-Scale. They consider an efficient solution as having a scalable model for archiving, efficient compression and indexing methods which should support a sufficiently expressive temporal query language. In order to expose such dynamic data at a Web-Scale, low-cost publication techniques like Triple Pattern Fragments (TPF) [9] could be used. Federated archive quering, which can not be done efficiently with current solutions, would become possible since the TPF framework supports this natively.

In this paper, we discuss the addition of versioning capabilities for instance-level changes to the lightweight TPF interface to reduce the publication and query cost of versioned Linked Open Data. This includes the requirements for such a solution and a research plan for future work.

2 Related Work

A survey about archiving Linked Open Data [3] categorizes instance-level archive solutions into three non-orthogonal storage strategies.

1. The *Independent copies* (IC) approach creates a separate instantiations of datasets for each change or set of changes.
2. The *Change-based* (CB) approach instead only stores changes between versions.
3. The *Timestamp-based* (TB) approach stores the temporal validity of facts.

Five foundational non-orthogonal query atoms for querying archiving systems were introduced [4].

1. *Version materialization* (VM) retrieves data using queries targeted at a single version.
2. *Delta materialization* (DM) retrieves query result differences between two versions.
3. *Version query* (VQ) annotates query results with the versions in which they are valid.
4. *Cross-version join* (CV) joins the results of two queries between versions.
5. *Change materialization* (CM) returns a list of versions in which a given query produces consecutively different results.

Triple Pattern Fragments (TPF) [9] is framework based on a REST interface for querying Linked Data. It was introduced as an alternative to expensive SPARQL endpoints. The TPF interface only allows single triple pattern queries, where linked pages contain the resulting triples. Full SPARQL queries are evaluated by clients, which split them into triple pattern queries, sending those to the server, and joining the results together locally.

3 Requirements

In order to add versioning support to TPF, we place requirements on the server interface, the server's internal storage model and the client that interacts with the interface.

3.1 Interface

The server interface influences the server load when publishing Linked Data archives. More complex interfaces lead to a higher number of possible requests and a lower level of cacheability. The TPF interface has parameters for triple pattern *subject*, *predicate* and *object*. In order to support the five foundational archive query atoms, this interface needs to be extended. It is however not required to support every one of these query atoms separately, since clients could derive some from others, at the cost of additional computation. For example, DM queries can be resolved using a VM interface by performing two VM queries and calculating the difference in results.

When extending a TPF interface for VM queries, we can expose a list of all available versions as reponse metadata or through a separate control. These versions could either directly link to interfaces at different versions [8] or those versions can be used as input to an extended interface with a *version* parameter. For example, a request for `s1 p1 ?` to version 5 using the second method could be `http://example.org?s=s1&p=p1&version =5`. DM queries similarly require a *version_initial* and *version_final* parameter to define the version range over which differences should be retrieved. A version query does not require direct changes to the interface, since all results are directly annotated with the versions in which they are valid [7]. CV queries need two times the set of parameters for VM queries, where the two sets define the queries to be joined server-side. CM requires a separate interface with parameters for a single triple pattern, which returns a list of versions for which the given triple pattern produces consecutively different results.

The choice of supported query atoms is a trade-off in server-client load, which can be illustrated using the Linked Data Fragments axis [9]. For a low client effort, it would be ideal for the server to support all five query atoms, leading to a higher server effort. If a low server effort is desired, only one of the first four query atoms should be supported, since the client can calculate any other query based on that.

3.2 Storage

Behind the server interface, there has to be a storage solution that is able to handle chosen query atoms. IC, CB and TB are three approaches for building such a storage solution.

Experimental results show that there exists a trade-off between the storage strategy, query efficiency and storage size [4]. If for example only VM queries are required, the IC policy might be selected if storage size is not an issue. Otherwise, the TB or a hybrid IC-CB approach might be more appropriate at the cost of an increase in server complexity.

Inspired by the classification of complexity for storage policies [3], Table 1 shows qualitative levels of complexity for each query atom. This shows that the supported query atoms at the TPF interface influence the complexity of the storage strategy.

3.3 Client

A TPF client should be able to understand the extended TPF server interface. For query atoms that are not supported on the server interface, the client should be able to simulate these using one or more supported query atoms. This simulation has a cost in terms of request count and computation, as shown in Tables 2 and 3. This shows that the choice of supported query atoms on the server will have an impact on the client query efficiency.

Users must be able to interact with one or more versions. This can be done by adding temporal query capabilities to the TPF client using archiving [6] or stream query languages [2]. Or by automatically selecting appropriate versions to query against.

	IC	CB	TB
VM	○	●	◑
DM	●	○	◑
VQ	◑	◑	○
CV	●	○	◑
CM	●	◑	◑

Table 1: Complexity (low ○, medium ◑, high ●) for evaluating **query atoms (rows)** on storage policies (columns).

	VM	DM	VQ	CV	CM
VM	1	n	1	1	
DM	2	1	1	2	
VQ	n	n	1	n	
CV	2	$2n$	1	1	
CM	n	n	1	n	1

Table 2: The number of queries required for evaluating **query atoms (rows)** using other query atoms (columns).

	VM	DM	VQ	CV	CM
VM	○	◑	◑	○	
DM	◑	○	◑	◑	
VQ	●	●	○	●	
CV	◑	●	◑	○	
CM	●	●	◑	●	○

Table 3: Complexity (low ○, medium ◑, high ●) for evaluating **query atoms (rows)** using other query atoms (columns).

4 Conclusions

To conclude, we foresee three tasks for supporting versioning in the TPF framework.

Task 1: Extension of the TPF interface for at least VM, DM, VQ or CV.

Task 2: A storage solution must be chosen depending on the required storage policies and query atoms.

Task 3: The TPF client must be able to consume the TPF interface extension for the desired and supported query atoms. Users can direct this behaviour either implicitly or through an expressive temporal query language.

These tasks entail a trade-off between server and client load, but also between server storage and server complexity.

Two TPF approaches already exists that meet to these requirements. The TPF Memento extension [8] supports VM queries, uses the IC storage policy and allows clients to select a target date for query evaluation. The TPF Query Streamer engine [7] supports VQ queries and stores data using the TB policy. It evaluates queries for a single moment in time.

No single solution will perform better than all other solutions in all cases. Therefore, we need to explore in which cases, which query atoms should be supported by the interface. And which storage policies and solutions are appropriate for the selected query atoms. It is also important to investigate how efficient query atoms can be simulated if only a subset is supported by the interface. A server and client cost model is needed to quantify the trade-off between server and client load, and between server load and storage. This cost model should have parameters for storage policies and the different supported client and server query atoms.

References

1. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1: Concepts and abstract syntax. Recommendation, W3C (Feb 2014), `http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`
2. Dell'Aglio, D., Della Valle, E., Calbimonte, J.P., Corcho, O.: RSP-QL semantics: a unifying query model to explain heterogeneity of rdf stream processing systems. International Journal on Semantic Web and Information Systems (IJSWIS) 10(4), 17–44 (2014)
3. Fernández, J.D., Polleres, A., Umbrich, J.: Towards efficient archiving of dynamic linked open data. Proc. of DIACHRON pp. 34–49 (2015)
4. Fernández, J.D., Umbrich, J., Polleres, A., Knuth, M.: Evaluating query and storage strategies for RDF archives
5. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 query language. Recommendation, W3C (Mar 2013), `http://www.w3.org/TR/2013/REC-sparql11-query-20130321/`
6. Meimaris, M., Papastefanatos, G., Viglas, S., Stavrakas, Y., Pateritsas, C., Anagnostopoulos, I.: A query language for multi-version data web archives (2015)
7. Taelman, R., Verborgh, R., Colpaert, P., Mannens, E., Van de Walle, R.: Continuously updating query results over real-time Linked Data. In: Proceedings of the 2nd Workshop on Managing the Evolution and Preservation of the Data Web (May 2016)
8. Verborgh, R.: Querying history with Linked Data (2016), `http://ruben.verborgh.org/blog/2016/06/22/querying-history-with-linked-data/`
9. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. Journal of Web Semantics 37–38, 184–206 (2016)