Towards Linked Data Update Notifications
Reviewing and Generalizing the sparqlPuSH approach

Magnus Knuth[1], Dinesh Reddy[1], Anastasia Dimou[2],
Sahar Vahdati[3], and George Kastrinakis[4]

[1] Hasso Plattner Institute – University of Potsdam, Potsdam, Germany,
magnus.knuth@hpi.de, dinesh.reddy@hpi.de
[2] Ghent University – iMinds, Ghent, Belgium,
anastasia.dimou@ugent.be
[3] Institute of Computer Science III – University of Bonn, Bonn, Germany,
vahdati@uni-bonn.de
[4] National Technical University of Athens, Athens, Greece,
george.kastrinakis91@gmail.com

Abstract. Linked Data resources change over time in terms of both content and relationships among them. Resources in a dataset might be frequently inserted, deleted, updated, and linked to other resources. Datasets are consumed in a lot of useful applications that would benefit from real-time notifications when data changes in RDF data stores. sparqlPuSH describes a notification service for updates in RDF stores. Analyzing this approach and the implementation for applicability, it became obvious that a number of serious constituent problems have not been addressed and remain unsolved. In this paper, we review sparqlPuSH approach and we introduce our own vision and ideas in extending and generalizing it.

Keywords: Linked Data, change notifications, RDF, SPARQL, update

1 Introduction

The Web of Data provides an enormous variety of content published to be (re)used in applications and interlinked with other datasets. Linked datasets change over time in terms of both links between resources and the content itself. Resources in a dataset might be frequently inserted, deleted, updated, and linked to other resources. The reasons for such updates can be manifold, e. g. availability of new data items, data quality improvements, agents feedback, etc. Therefore, Linked Data consumers need to be informed with real-time notifications when data changes in external RDF data stores.

This paper acknowledges the work of Passant and Mendes [14] while showing the shortcomings that make their solution a hardly generalizable artifact. Besides highlighting the inherent problems, we suggest generalizable solutions, whereas we conclude that some problems call for compromise solutions depending on the dataset characteristics.

At first we describe our motivation towards linked data notifications in Section 2. Then in Section 3 we briefly explain the sparqlPuSH approach, benefits of using SPARQL, and Push vs. Pull notification mechanisms. Further in Section 4, we discuss preliminaries and shortcomings of the sparqlPuSH approach. Furthermore, we shortlist the requirements to overcome shortcomings of the sparqlPuSH approach and discuss existing open research problems in Section 5 and 6 respectively. Finally in Section 7, we summarize our conclusions and future work.

2 Motivation

In the vast Web 2.0, it is rather difficult to get live updates or notifications about the information we want at more concrete level, for instance about a particular concept or event. One might be interested in following a particular topic (e. g. Greek Elections), or getting live updates about a particular company from stock market. Unfortunately, one can get notified through e-mails and RSS feeds only at abstract level, e. g. about recent news articles related to these concepts, but not *real-time* and on a *concrete level*, e. g. about a particular resource description. This occurs because it is still not possible to take advantage of valuable information hidden in articles in an automated way, and thus, users still have to spend lot of time to distinguish the information they need from the retrieved resource.

Taking advantage of Semantic Web technologies we aim to overcome this problem. From the information retrieval point of view, in comparison to traditional keyword search, SPARQL queries provide a more expressive means to describe user's information needs. There are a number of Linked Data based services that would benefit from user notifications at concrete level:

- *Personal notifications* regarding new/updated data of interest, such as real estate offers, product and product price information, or upcoming conferences, new interesting papers, and new citations of own papers.
- *Cache invalidation* for applications that temporarily keep copies of external data for reduced data access times.
- *Interlinked Datasets* Linked Data applications typically use multiple sources. Often these data sources are updated, affecting existing links which might end up being broken or resource URI's might be updated. Notifying subscribers on time about updated links, such that they, in their turn, update these dataset links respectively.

Currently, the pioneer Linked Data notification approach, sparqlPuSH, allows users to register with a SPARQL query and get updates pushed to the user as new matching triples arrive in an underlying triple store containing relevant data. However, the sparqlPuSH implementation presents limitations (c. f. Section 4.2) that make it not applicable in all of the aforementioned cases and not able to deal with the constantly evolving RDF data available at Web scale.

3 *sparqlPuSH* approach

The *sparqlPuSH approach* [14] describes a notification service for updates in RDF stores. It relies on SPARQL queries, tracks changes of the result set, published as an RSS feed, and broadcasts change notifications via the PubSubHubbub protocol [5]. It allows any application with a SPARQL endpoint to broadcasts change notifications of RDF data in real-time. Once a user registered a SPARQL query related to his information need, he gets notified when data within the store related to that query changes, i. e. when the result set varies compared to the previous state.

3.1 Defining information needs using SPARQL

Defining a SPARQL query is a rather expressive way to express an information need against an RDF dataset. It allows to construct and filter arbitrary graph patterns. SPARQL provides four different query forms: `SELECT`, `ASK`, `DESCRIBE`, and `CONSTRUCT` which return solutions in the form of result sets or RDF graphs providing information according to the user's needs. For illustration, we list a few SPARQL queries which can likely be used for notification.

(Q1) *Specific resource characteristics*, e. g. when the president of Italy changes.
```
SELECT ?pres WHERE { :Italy :leader ?pres . }
```
(Q2) *Information that is not present yet*, e.ġ. notify when there is a female president of the United States.
```
SELECT ?pres WHERE { :United_States :leader ?pres .
  ?pres :gender :Female . }
```
(Q3) *Signaling a status change*, e. g. notify when the proceedings of a particular workshop got published.
```
ASK { :NoISE15 :proceedings ?proc .
  ?proc :publishedBy ?publisher . }
```
(Q4) *Monitoring individual resource descriptions*, e. g. notify when the resource of the city of Berlin got updated.
```
DESCRIBE <http://dbpedia.org/resource/Berlin>
```
(Q5) *Construction of new triples*, e. g. notify when new resources with the same unique identifier (inverse functional property) pop up in the dataset.
```
CONSTRUCT { ?a owl:sameAs ?b } WHERE { ?a :hasUniqueID ?id .
  ?b :hasUniqueID ?id . FILTER (?a != ?b) }
```

3.2 Push (notification) vs. Pull (polling)

There are two distinct ways to inform consumers about data changes.

- *Pull mechanisms* demand the user to poll a resource, such as an RSS feed, frequently in order to detect an update, that might be useful to downsize complex requests. But consumers are not informed immediately when data changes.

– *Push mechanisms*, such as remote procedure calls (RPC) and webhooks, notify the consumer proactively and reduce the amount of requests in an efficient way.

Both approaches are eligible and should be supported. *PubSubHubbub* defines a scalable mechanism to do so. It is a protocol based on the Atom model of exposing services by feeds, extending Atom's pull mechanics with a Publish-Subscribe mechanism. It allows clients to subscribe callbacks with "hubs". Whenever a feed gets updated, the clients will be notified through their callbacks [5].

4 Preliminaries

4.1 State of the Art

The study of Linked Data notifications is very relevant for a broad range of application domains. Earlier studies related to detecting changes in the RDF data are: *DSNotify* [6], a generic framework introduced to fix broken links between different datasources and a datasource itself. *Resource SubscrIption and Notification sErvice (rsine)* [11] is a framework that notifies subscribers whenever resources are updated, created, or removed. It is comparable to sparqlPuSH but is designed to operate on a more general level. In contrast to sparqlPuSH, *rsine* intends to maintain quality of controlled vocabularies. *Boca RDF* [12] provides a change detection sub-system based on Sun's Java messaging service. Users will get notifications while there is any change in single RDF statements or the entire graph. *PingTheSemanticWeb*[5] (*PTSW* in short) provides notification services about recently created or changed RDF documents. It offeres XML-RPC and REST APIs pointing to the time and location of the latest updated RDF Data Source.

4.2 *sparqlPuSH* Shortcomings

Even though the sparqlPuSH *approach* is well-grounded, the existing sparqlPuSH *implementation* is rather limited to a certain use case. The *implementation* is intended to be used for micro-blogging notifications and can not be generalized for broader but common user information needs.

Firstly, the presented implementation of sparqlPuSH imposes strong restrictions on SPARQL queries:
(1) Only `SELECT` queries are allowed.
(2) The query should contain a `?uri` and a `?date` variable in the `SELECT` clause.
(3) Beyond that, only `?label` and `?author` variables are allowed in the result.

These restrictions work for the selected micro-blogging use-case but allow only a very limited application. Even though the authors claim their system "can be plugged on top of any SPARQL endpoint", it is not possible to generalize the approach in order to fit common information needs on different datasets.

[5] `http://pingthesemanticweb.com`

Furthermore, these restrictions mask out the actual difficulties that come along with allowing arbitrary queries. While the `?date` variable allows to identify a modification by simply keeping the last modification date, typical queries do not have such an indicator per sé. A generic solution should not rely on the values of the variables of the result set to compare two subsequent versions, but any returned result set should be compared with the latest results derived from the SPARQL endpoint.

Secondly, the presented implementation demands updates to be done via the sparqlPuSH interface itself, in order to trigger change events. It is therefore limited to SPARQL endpoints that are under full control of the notification service provider. Depending on the RDF store, changes can typically be made via multiple update methods. E. g., the OpenLink Virtuoso triplestore additionally provides a Conductor UI and a JDBC/ODBC compliant ISQL interface. RDB2RDF servers often do not provide a SPARQL Update compliant endpoint, because they work as one-way RDF exporters. Detecting change events is crucial for knowing when to re-evaluate (which) registered queries and triggers are typically not available.

5 Requirements

Based on the original sparqlPuSH approach and in order to address the limitations of the case-specific implementation, we summarize the requirements and propose generic solutions, considering the current state of the art.

(R1) **Processing arbitrary SPARQL queries**
The service should be able to support all types of SPARQL queries. Since SPARQL queries is based around graph pattern matching, both basic graph patterns and any form of group graph patterns should be covered.

(R2) **Application on any accessible SPARQL interface**
The service should be able to work on top of any SPARQL interface, including any public external endpoint, as well as internal endpoints or any other SPARQL interface, e. g. a client of Triple Pattern Fragments[6] or a query directly executed against a file with data in RDF.

(R3) **Avoidance of unnecessary load to SPARQL interfaces**
The service should avoid any unnecessary load towards the utilized SPARQL interfaces, i. e. queries should only be re-evaluated when a change can be assumed, data transmission should be minimized and requests delayed.

(R4) **Sufficiently expressive description of what has changed**
Within the feed a description of the change should be given, that allows the agent to determine the relevance of a change.

[6] `http://client.linkeddatafragments.org/`

6 Open Research Problems

Derived from the imposed requirements R1 to R4 and the assessment of the sparqlPuSH implementation, we identified a number of constituent problems that are currently unsolved.

(P1) **Handling large SPARQL query results** (R1, R2, R3)
(P2) **Comparison of SPARQL query results** (R1)
(P3) **Scheduling the re-evaluation of SPARQL queries** (R3)
(P4) **Equality of SPARQL queries** (R3)
(P5) **Describing changes in SPARQL query results** (R4)

6.1 Handling large SPARQL result sets

The main disadvantage of using SPARQL queries for detecting changes within a dataset is that results can get enormously huge. In order to compare the current result set of a query with another one in the future, the complete results have to be retrieved and sufficient information about the current result has to be stored. E. g. a query for all known redirects on DBpedia 2014 returns a result set with 6,473,988 rows, equaling ~650 MByte serialized as TSV and ~1.3 GByte as XML:

```
SELECT ?a ?b WHERE { ?a dbo:wikiPageRedirects ?b . }
```

Moreover, public SPARQL endpoints often do not return the complete result set. For performance reasons their result set size is typically limited, e. g. to a value of 10,000 rows [3]. To get the complete result set, this needs to be circumvented technically, e. g. by paginated requests[7]. Thus, one challenge is to keep the amount of data that needs to be transmitted and stored as low as possible. Storing and transmitting the whole result set may be very expensive for particular queries.

We propose the following research question: *How can the SPARQL result size be limited efficiently without losing relevant information for change detection?*

Aggregation One solution would be to request (and store) aggregates about SPARQL queries. Aggregates are typically way smaller but only deliver incomplete information. A given SPARQL `SELECT` query can be rewritten in order to retrieve only the number of results (result set rows) and only store that number instead of the original result: `SELECT COUNT(*) WHERE ...` This number can be compared with the number of results of a following result. This allows to find changes to the data where the number of result sets differs, i. e. we miss those changes where the number of results remains equal while the content changes. Nevertheless, this approach fits well for datasets that data is exclusively added.

Similarly, a query could be rewritten to return a minimum, maximum, or average value for a particular variable. For minimum and maximum values, we

[7] E. g. by using `QueryExecutionFactoryPaginated` from the Jena SPARQL API: `https://github.com/AKSW/jena-sparql-api`

have to assume that data would be added or removed in a particular order on a particular variable, e. g. `SELECT MAX(?releaseDate) WHERE ...` could work for a dataset containing news items which have a publishing date. Certainly, it would be necessary to understand the characteristics of the dataset and the semantics of the query in order to rewrite a query automatically in such a way.

When using aggregates it might be possible to compare result sets for a change, though the change can only be described in an equally aggregated from.

Hashing A common way to compare large amounts of data is to create hash values for the data, store the hashes and later if the data needs to be compared, just compare the hashes. As hash values typically have a length less than 128 Byte the amount of data to store can be reduced. The probability that two different result sets for the same query produce equal hash values is extremely low as long as the serialization format and the order of the results remain the same. By using hash values it is possible to compare result sets for equality, though it can not be concluded in which way or to which extend the result set changed.

The hash functions built in SPARQL 1.1[8] can be used on the result set solution level, in order to compact large-size RDF terms.

Streaming An alternative approach to compare large result sets, that lately gains ground, rely on streaming the result sets of SPARQL queries. Triple Pattern Fragments servers is such a solution that resolve queries for basic triple patterns while Triple Pattern Fragment clients resolve queries of any type of group patterns.

6.2 Comparison of SPARQL query results

Another challenge is to guarantee a high reliability of the comparison method, i. e. changes should be reported if and only if the result set really changed.

We propose the following research question: *How can the SPARQL results be effectively and efficiently compared?*

ASK queries are considered to test whether or not a query pattern has a solution. Such queries are ideal for queries that no information is expected to be returned about the possible result set, just whether or not a solution exists (e. g. Q3). Comparison of boolean results for `ASK` queries is trivial.

DESCRIBE and CONSTRUCT queries RDF graphs being the result of `DESCRIBE` and `CONSTRUCT` can be compared by usual RDF Diff implementations[9].

The main problem of detecting graph differences is the canonical labelling of blank nodes, which is an issue of the graph isomorphism problem [16]. Ongoing research addresses these issue, whereas [7] seems promising.

[8] `http://www.w3.org/TR/sparql11-query/#func-hash`
[9] `http://www.w3.org/2001/sw/wiki/How_to_diff_RDF`

SELECT queries For comparison of SPARQL `SELECT` query result sets it needs to be distinguished between ordered and unordered result sets. For ordered result sets each row in the result set needs to match the row with identical index in the other result set. While for unordered result sets each row of the result set has a matching row in the other result set.

The Jena ARQ API[10] provides an implementation of `ResultSetCompare` which allows to check equivalence of result sets either by value or order. The implementations return only a boolean result and don't provide an analysis of the difference between the result sets, which would be beneficial for change descriptions. Internally the result sets after some elementary checks are transformed to RDF graphs using the *W3C result set vocabulary*[11] and then these graphs are compared for equivalence. Here, the same problem of graph isomorphism applies and blank nodes in result sets should be avoided.

In the case of Triple Pattern Fragments, a first comparison can rely on the metadata. If the total number of triples count has changed, it indicates that the result set of triples of a certain graph pattern has changed. If the number of total triples remains the same, processing of graph patterns is required at the client side based on the results returned from the server. A change in one of the requested patterns indicates a (possible) change in the final result set.

6.3 Scheduling

The third challenge is to evaluate the best time to perform the next check for updates. The simplest solution would be to revalidate all queries at defined time intervals or using a round-robin (RR) appproach. But since such a check is costly and produces unnecessary load to the SPARQL interface if there was no update, it should be performed only when a relevant change is likely to have happened.

The research question is: *How to determine the best time at which a relevant data change has occured?*

Dataset Descriptions Datasets have varying characteristics concerning their update frequency, e. g. DBpedia is usually updated once a year while DBpedia Live resources constantly change as soon as relevant changes have been made to the respective Wikipedia article or the DBpedia Mappings.

The VoID vocabulary [1] enables descriptions of an RDF dataset's characteristics. It contains concepts to describe general metadata (e. g. licenses, author), access metadata (e. g. SPARQL endpoint, data dump URI, lookup URIs), and structural metadata (e. g. patterns, partitionings, vocabularies statistics). In addition, one can describe the relations with other datasets using a *linkset*. The `void:triples`, `void:entities` and `void:documents`, `void:distinctObjects`, or `void:distinctSubjects` could be considered to assess if the result set of a SPARQL query has changed. While those numbers could act as an indicator of changes, combinations of updates could result in the same numbers.

[10] `http://jena.apache.org/documentation/query/`
[11] `http://www.w3.org/2001/sw/DataAccess/tests/result-set`

The DCAT vocabulary[12] is the W3C standard to be used for the description of data catalogs. Data catalogs are centralized indexes or repositories that contain dataset metadata. DCAT contains high-level metadata for such catalogs (e. g. title, licenses, version) and datasets (e. g. keywords, language). Among other properties, the DCAT recommendation proposed the use of `dct:date`, `dct:accrualPeriodicity`, `dct:created`, `dcterms:issued`, and `dct:modified` properties to describe the metadata of a dataset and in our case for the result set. If the modification date of a dataset is at a later time slot than the last query execution, the result set for the queries for this dataset should be re-evaluated.

Sensoring updates Dataset publishers could provide a dataset update notification process by themselves in order to trigger a query re-evaluation.

In the use case of DBpedia Live, updates to the dataset occur almost continuously and the actual changesets (updates in form of inserted and removed triples) are available. Still it is not trivial to compute the necessity of re-evaluation of particular SPARQL queries based on this changesets.

Estimating update intervals If there is no reliable modification data available for a dataset, update intervals could be learned by the system. A record of changes on a dataset would indicate when the next update might occur.

Simple estimations on cached object freshness are implemented in web proxy server software, such as the *Apache Traffic Server*[13]. The freshness limit of an HTTP object is typically determined based on the time interval of the most recent modification. As the freshness limit of a query would exceed, it is needed to be revalidated. The scheduling adapts to the change characteristics of the query, as in case the result set is still fresh, the next validation will be scheduled with an increased freshness limit.

6.4 Equality of SPARQL queries

With an increasing number of registered SPARQL queries, it becomes likely that duplicates occur. Different queries might meet the same information need and return the same results. Moreover, there are unlimited possibilities to express the same query. Regarding a SPARQL query as a string of characters, a simple change of binding names or prefix definitions, a varying order and the manifold possibilities for abbreviated notation of basic graph patterns, alternative property path expressions, optional whitespaces, etc. will lead to unequal queries. Beyond syntax variations, completely different queries could behave exactly the same on a particular dataset, as e. g. when the dataset contains equivalent classes and queries ask for another one of these classes in each case.

In order to avoid such duplicate queries, it is necessary to match queries with those already registered. A user could be suggested to re-use an already existing

[12] `http://www.w3.org/TR/vocab-dcat/`
[13] `http://trafficserver.apache.org/`

query, or the query and respectively the query results could be rewritten to fit an equivalent query transparently to the user. In order to ensure the soundness and completeness of the rewriting, the results of the existing query must be able to be used to produce the same results as executing the original query.

Dividino and Gröner summarize the efforts on detecting equal or similar SPARQL queries [4] and formulate the research question: *Which of the following SPARQL queries are similar? Why?*.

Syntactical Query Similarity Syntax variations could be eliminated by transforming queries to their canonical form. There have been efforts towards normalizing graph pattern [15] and fragments of the SPARQL language [9], to our knowledge there is no canonical form for general SPARQL queries so far. Syntactical similarity approaches often rely on the Levenshtein distance [10, 13] either on the whole query string or parts, such as the triple patterns. However, a small Levenshtein distance does not necessarily indicate that queries are semantically related or represent equal information needs. For this reason these approaches seem inappropriate, at least for detecting equal queries.

Structural Query Similarity Query rewriting applications commonly use similarity measures based on graph matching. In essence, queries are represented as a graph and the goal is to find the maximum common sub-graphs among such query graphs. Le et al. define a similarity metric representing the structural overlap of two queries and propose an efficient algorithm for query rewriting of simple queries (i. e. without `FILTER`) [8]. Letelier et al. transform the SPARQL fragment of well-defined queries into pattern trees and based on that provide testing of query equivalence and containment [9].

6.5 Describing changes in SPARQL query results

Within the feed a description of the change should be given in a formal way, that allows an agent to determine the relevance of a change in his context. It is currently unclear, what information a user would consider relevant. In many cases it might be sufficient to be notified about any change, since running the local update process might be less expensive, than evaluating the implications of a particular change.

We formulate the following research question: *How can changes of SPARQL query results be described in an extent useful to the end user?*

Result set change as RDF change Not only the result sets returned for the same queries are of interest to be explicitely described, but also the transition from the one result set to the other. To describe a result set change in full depth, all alteration must be described.

To this end, SPARQL `SELECT` query result sets should be serialized as RDF using the *W3C result set vocabulary*. This allows to describe result set changes as

RDF changes. Another advantage of this approach is that changes of `DESCRIBE` and `CONSTRUCT` queries can be described in the same or comparable way.

There are a number of schemas defined to describe RDF changes:

Delta ontology [2] tries to uniquely identify what is changing and to distinguish between the pieces added and removed.

Changesets[14] defines a set of terms for describing changes to resource descriptions. The vocabulary introduces the notion of a ChangeSet which encapsulates the delta between two versions of a resource description which is represented by two sets of triples: additions and removals.

Graph Update Ontology[15] (GUO) is aiming at enabling lightweight RDF graph updates and graph synchronisation per triple level. GUO tries to complement SPARQL UPDATE without the need for Quads (as in TriG or TRiX) or Reification (as in Changesets).

RDF Patch[16] is a file format for recording changes made to an RDF dataset and can be used for replicating changes between multiple copies of the same dataset, in this case of the result set. The changes are recorded considering the triples which are added or deleted to the default graph, and the quad for a named graph.

7 Conclusions

In this paper we identified requirements for an effective change notification system for the Web of Data. We evaluated existing approaches for notifications with the most prominent approach, sparqlPuSH, that currently exists, showing only a limited functionality but acting though as potential demonstrator. We identified issues that need to be tackled in order to achieve the desired functionality in a generic fashion. These issues are non-trivial problems, which should be targeted by future research. Regarding performance and scalability, we concluded that there might not be an ultimate solution for these problems, different solutions rather depend on the given setting and should be evaluated as such. We initiated an implementation of the described notification system, which is thought to be a generic framework for covering multiple solutions for the named issues.

8 Acknowledgements

The work presented is part of the author's hackaton project for the Web Intelligence Summer School 2014[17].

[14] `http://vocab.org/changeset/schema.html`
[15] `http://webr3.org/specs/guo/`
[16] `http://afs.github.io/rdf-patch/`
[17] `http://www.emse.fr/~zimmermann/WI_2014_Site/`

References

1. Alexander, K., Cyganiak, R., Hausenblas, M., Zhao, J.: Describing linked datasets - on the design and usage of VoID, the 'vocabulary of interlinked datasets'. In: Linked Data on the Web. Madrid, Spain (2009)
2. Berners-Lee, T., Connolly, D.: Delta: an ontology for the distribution of differences between RDF graphs (2004)
3. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.Y.: SPARQL web-querying infrastructure: Ready for action? In: The Semantic Web – ISWC 2013, Lecture Notes in Computer Science, vol. 8219, pp. 277–293. Springer (2013)
4. Dividino, R., Gröner, G.: Which of the following SPARQL queries are similar? why? In: 1st International Workshop on Linked Data for Information Extraction. CEUR Workshop Proceedings, vol. 1057. CEUR-WS.org (2013)
5. Fitzpatrick, B., Slatkin, B., Atkins, M.: PubSubHubbub core 0.3–working draft. Project Hosting on Google Code (2010)
6. Haslhofer, B., Popitsch, N.: DSNotify - detecting and fixing broken links in linked datasets. In: 20th International Conference on Database and Expert Systems Application – DEXA 2009. pp. 89–93. IEEE (2009)
7. Hogan, A.: Skolemising blank nodes while preserving isomorphism. In: 24th International World Wide Web Conference – WWW 2015. ACM (2015)
8. Le, W., Kementsietsidis, A., Duan, S., Li, F.: Scalable multi-query optimization for SPARQL. In: 28th International Conference on Data Engineering – ICDE. pp. 666–677. IEEE (2012)
9. Letelier, A., Pérez, J., Pichler, R., Skritek, S.: Static analysis and optimization of semantic web queries. In: Proceedings of the 31st Symposium on Principles of Database Systems. pp. 89–100. PODS '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2213556.2213572
10. Lorey, J., Naumann, F.: Detecting SPARQL query templates for data prefetching. In: The Semantic Web: Semantics and Big Data – ESWC, Lecture Notes in Computer Science, vol. 7882, pp. 124–139. Springer (2013)
11. Mader, C., Martin, M., Stadler, C.: Facilitating the exploration and visualization of linked data. In: Linked Open Data – Creating Knowledge Out of Interlinked Data, pp. 90–107. Springer (2014)
12. Missier, P., Alper, P., Corcho, Ó., Dunlop, I., Goble, C.: Requirements and services for metadata management. IEEE Internet Computing 11(5), 17–25 (2007)
13. Morsey, M., Lehmann, J., Auer, S., Ngomo, A.C.N.: Usage-centric benchmarking of RDF triple stores. In: AAAI (2012)
14. Passant, A., Mendes, P.N.: sparqlPuSH: Proactive notification of data updates in RDF stores using PubSubHubbub. In: SFSW. CEUR Workshop Proceedings, vol. 699. CEUR-WS.org (2010)
15. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. In: The Semantic Web – ISWC 2006, pp. 30–43. Springer (2006)
16. Tzitzikas, Y., Lantzaki, C., Zeginis, D.: Blank node matching and RDF/S comparison functions. In: The Semantic Web – ISWC 2012. Lecture Notes in Computer Science, vol. 7649, pp. 591–607. Springer (2012)