SCRY: extending SPARQL with custom data processing methods for the life sciences

Bas Stringer[1,3], Albert Meroño-Peñuela[2], Sanne Abeln[1], Frank van Harmelen[2], and Jaap Heringa[1]

[1] Centre for Integrative Bioinformatics (IBIVU), Vrije Universiteit Amsterdam, NL
[2] Knowledge Representation and Reasoning Group, Vrije Universiteit Amsterdam, NL
[3] To whom correspondence should be adressed (`b.stringer@vu.nl`)

Abstract. An ever-growing amount of life science databases are (partially) exposed as RDF graphs (e.g. UniProt, TCGA, DisGeNET, Human Protein Atlas), complementing traditional methods to disseminate biodata. The SPARQL query language provides a powerful tool to rapidly retrieve and integrate this data. However, the inability to incorporate custom data processing methods in SPARQL queries inhibits its application in many life science use cases. It should take far less effort to integrate data processing methods, such as BLAST, with SPARQL. We propose an effective framework for extending SPARQL with custom methods should fulfill four key requirements: generality, reusability, interoperability and scalability. We present **SCRY**, the **S**PARQL **c**ompatible se**r**vice la**y**er, which provides custom data processing within SPARQL queries. SCRY is a lightweight SPARQL endpoint that interprets parts of basic graph patterns as input for user defined procedures, generating an RDF graph against which the query is resolved on-demand. SCRY's federation-oriented design allows for easy integration with existing endpoints, extending SPARQL's functionality to include custom data processing methods in a decoupled, standards-compliant, tool independent manner. We demonstrate the power of this approach by performing statistical analysis of a benchmark, and by incorporating BLAST in a query which simultaneously finds the tissues expressing Hemoglobin β and its homologs.

Keywords: SPARQL, customization, extension, RDF generation, data processing

1 Introduction

The Semantic Web continues to grow, reaching ever more knowledge areas and scientific communities[17]. Life sciences are adapting to this technology[26], and an ever-increasing number of databases are (partially) exposed as RDF graphs (e.g. UniProt[13], TCGA[16], DisGeNET[12], Human Protein Atlas[23]), complementing traditional methods to disseminate biodata.

The SPARQL query language can be used to express queries across these diverse data sources in a universal manner [22]. SPARQL queries work on RDF representations of data by finding patterns that match templates in the query, in effect finding information graphs in the RDF data based on the templates and filters (constraints on nodes and edges) expressed in the query. Such patterns match relations that are explicitly present in the RDF graph, or that are derivable from the graph under a given entailment regime such as RDFS or OWL-DL [21]. However, there are many cases where information needs to be retrieved from an RDF graph where it is (a) impractical to store all relevant relation-instances explicitly in the graph, and (b) the relevant relation-instances can not be derived from the explicit graph under any of the supported entailment regimes.

A straightforward example of such a problem, is selecting outliers from a set of query results. Which results are outliers depends on a scoring function and the rest of the set. It is infeasible to precompute which results are outliers for every putative result set and scoring function, and the math for typical scoring functions is not natively supported by SPARQL or RDF/OWL entailment.

Another example, common in life science, involves finding proteins which are evolutionarily related to a given protein of interest. There is a multitude of methods and parameters through which this can be derived, but which of these should be used strongly depend on the exact context of the question. Precomputing every possible relations between all proteins is not only impractical, but combinatorially impossible. Again, RDF/OWL entailment can not derive such complex relations ad hoc.

These examples illustrate that, even though RDF has shown itself to be a very versatile representation language, there are important relations between entities which can in principle be expressed in RDF, but that are impractical or impossible to materialize persistently in a triplestore. Hence, they are not accessible through SPARQL queries over such a triplestore. At the same time, these relations are critical to resolve queries common to the life sciences or other domains.

We present the **S**PARQL **c**ompatible se**r**vice la**y**er (SCRY) as a solution to this problem. SCRY is a lightweight SPARQL endpoint that allows users to create their own procedures, assign them to a URI, and embed them in standard SPARQL queries. Through these procedures, analytic methods can be executed at query time, effectively extending SPARQL with API-like functionality. SCRY provides a framework for custom data processing within the context of a Semantic Web query, facilitating on-demand computation of domain-specific relations in a generalized, reusable, interoperable and scalable manner.

2 Requirements

As introduced above, RDF/OWL entailment is not sufficiently expressive to incorporate complex domain-specific data processing methods in SPARQL queries. We observe there is an unmet need for a platform which enables computational biologists to efficiently implement and integrate such methods. A platform enabling this would ideally be 1) easy to deploy, 2) easy to extend with new methods, and 3) easy to write queries for. Concretely, we argue this can be achieved by meeting the following requirements:

Generality A generic platform, which enables the implementation of any method from any domain, avoids Semantic Web services devolving to a multitude of mutually incompatible "SPARQL-like" extensions (e.g. [2]). Consequently, the interface through which implemented methods are exposed should support arbitrarily complex algorithms, maintaining SPARQL's expressivity without restricting functional extensibility.

Reusability Implemented methods should be standardized to a degree that allows them to be reused between queries. Done well, this facilitates sharing services and reusing the work of others, effectively extending the shareable nature of Semantic Web data and schemata to apply to methods as well.

Interoperability Queries incorporating methods which extend SPARQL's functionality should maintain full compatibility with the SPARQL standard, such that any SPARQL-aware server, service or tool can parse and resolve them. Towards this end, methods must be implemented in an endpoint-independent manner.

Scalability Many analytic methods are computationally expensive, making them inhibitively slow or outright impossible to run through remotely hosted services. Simpler methods incur bandwidth restrictions if they are given high numbers of inputs, or produce large volume outputs. Such methods benefit from using local resources for computation, which requires users to control both the software and the hardware resolving their queries.

3 Related Work

The simplest and most prevalent way of processing RDF data accessed via SPARQL endpoints is post hoc scripting. This typically involves SPARQL-compatible libraries in some general purpose programming language [6,9]. Despite its generality, this comes with two pitfalls. Firstly, such libraries are highly coupled with the standards of SPARQL and the programming languages they are implemented in, making their maintenance costly. Users have to implement purpose-specific data processing pipelines, which is time consuming and typically results in code that is not reusable [7]. In fact, post hoc scripting does not functionally extend SPARQL as a query language, and thus lacks interoperability and reusability.

The SPARQL 1.1 specification [25] defines Extensible Value Testing (EVT) as a mechanism to extend SPARQL's standard functionality. However, custom EVT procedures are restricted to appear in limited query environments like `BIND()` and `FILTER()`), thus obstructing the customization of aggregates. Moreover, queries "using extension functions are likely to have limited interoperability" [25], making them triplestore dependent and forcing redundant implementations across products. Another take on extending SPARQL is OGC GeoSPARQL, which "defines a vocabulary for representing geospatial data in RDF, and defines an extension to the SPARQL query language for processing geospatial data" [2]. However, its reliance on endpoint customization impairs interoperability.

Web Services are a common way of exposing data processing functionality online. Significant efforts have been made to describe Web Services semantically to enhance their interoperability [4]. For example, in [19] authors propose to expose REST APIs as Linked Data. In the same line of thought, the Semantic Automated Discovery and Integration (SADI) registry [27] uses an elegant OWL description of the inputs and outputs of semantic services, which enables non-expert users to automatically incorporate relevant methods within their queries. However, SADI's dependence on compatible

	Generality	Reusability	Interoperability	Scalability
Post hoc scripting	✓			✓
SPARQL extension standards (EVT, GeoSPARQL)		✓		
Semantic Web Services (SADI)	✓	✓		
Linked Data APIs (OpenPHACTS)		✓		✓

Table 1: Reviewed work versus met requirements.

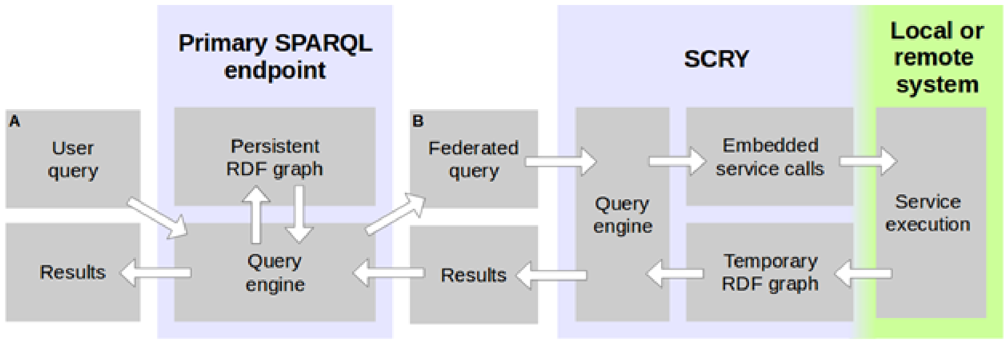


Fig. 1: Dataflow diagram typical queries using SCRY: the user submits query **A** to their primary endpoint of choice, with a `SERVICE` clause indicating part of the query should be federated to an instance of SCRY. SCRY in turn parses embedded service calls out of the federated query, whose (local or remote) execution populates the RDF graph containing the query's solution. The federated query is resolved as if SCRY were a traditional SPARQL endpoint, and results are returned to the federating side where they can be returned directly to the user, or used for further graph evaluation. Alternatively, users can interact with a SCRY instance directly by submitting query **B** themselves, instead of federating it through a primary endpoint.

query engines (e.g. SHARE [24]) limits interoperability, and its scalability suffers from registered services commonly being exposed exclusively through online, single-point-of-failure endpoints.

Linked Data APIs are very similar to Web Services in terms of their goals and challenges [11]. They expose Linked Data in Web standard formats without requiring extensive knowledge of RDF or SPARQL [5,14]. Projects like Open PHACTS [28] maintain runtime performance by restricting query expressivity, trading in generality for scalability. Given sufficient documentation, Linked Data APIs can be very reusable, but since they abstract SPARQL away from end users, they are not directly interoperable.

Table 1 summarizes which of our requirements from the previous section are fulfilled by the listed approaches.

4 SCRY

Infrastructure: Our **S**PARQL **c**ompatible se**r**vice la**y**er (SCRY) acts as a lightweight SPARQL endpoint, granting users access to a personalized set of services during query execution. Through these services, SCRY allows users to incorporate algorithms of arbitrary complexity within standards-compliant SPARQL queries, and to use the generated outputs directly within those same queries. Unlike traditional SPARQL endpoints, the RDF graph against which SCRY resolves its queries is generated at query time, by executing services encoded in the query's graph patterns.

Every service exposed by SCRY comprises one or more procedures. Registered procedures are associated with a URI, which SCRY will recognize when it receives a query. These procedure-associated URIs, as well as the desired inputs and outputs, can be embedded within the graph patterns of standards-compliant, syntactically pure SPARQL queries (see figure 2). Prior to resolving the SPARQL query, SCRY executes embedded procedures and adds their output to the queried RDF graph. Procedures can comprise simple instructions, like rounding off a decimal number. They can also be arbitrarily complex, for example running an external program, parsing its output, performing some form of analysis and exposing specifically requested results as RDF. The output of one procedure can be used as input for the next, allowing simple (non-circular) workflows to be encoded.

Figure 1 illustrates the two typical dataflows of SPARQL queries resolved through SCRY, accessing it through **(A)** a federated query from a primary endpoint or **(B)** a direct query to a live SCRY instance.

Engineering effort: To ensure SCRY can be easily utilized by life scientists, we aimed to minimize the effort needed to 1) deploy SCRY, 2) extend it with novel data processing methods and 3) write queries invoking them.

SCRY can be deployed on any system running Python 2.7. The framework itself, as well as the services required for the use cases we describe below, can be easily installed using the package manager `pip`.[4] It is equally straightforward to extend SCRY with services of your own. Python scripts added to SCRY's working directory are automatically checked for defined procedures, which can take as little as three lines to define.[5] SCRY can be further configured through a commandline interface, which includes options to load services from elsewhere.

Defining a procedure requires 1) a URI and 2) a Python function to execute when SCRY finds this URI in a graph pattern. Note these functions can rely on local or remote resources, including other Python functions, shell commands (e.g. through `os.system()`), and web requests.

Figure 2 shows how easily procedure calls can be embedded in SPARQL queries using SCRY. We explain the query itself, and how SCRY interprets the embedded procedure calls, in the Statistics use case of the Evaluation section.

5 Evaluation

We evaluate whether or not SCRY meets the requirements described in section 2 based on two use cases, from the domains of statistics and bioinformatics respectively.

[4]See `http://tinyurl.com/scry4ls#file-deployment`

[5]See `http://tinyurl.com/scry4ls#file-hello-world`

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX scry: <http://scry.rocks/>
PREFIX math: <http://scry.rocks/math/>
PREFIX in:   <http://scry.rocks/input?>
PREFIX out:  <http://scry.rocks/output?>

SELECT ?vendor ?count ?mean ?median ?sd {
  SERVICE <http://bsbm.endpoint.here> {
    SELECT ?vendor
           (COUNT(?review) AS ?count)
           (AVG(?rating) AS ?mean)
           (GROUP_CONCAT(?rating; SEPARATOR=',') AS ?ratings)
    WHERE {
      ?review bsbm:reviewFor ?product ;
              bsbm:rating1   ?rating .
      ?offer bsbm:product ?product ;
             bsbm:vendor  ?vendor .
    } GROUP BY ?vendor}
  GRAPH ?g1 {math:median in: ?ratings; out: ?median .}
  GRAPH ?g2 {math:stdev  in: ?ratings; out: ?sd     .}
} ORDER BY ?vendor
```

Fig. 2: Query for the statistics use case, utilizing SCRY to calculate standard deviations on simulated e-commerce data from the Berlin SPARQL Benchmark.

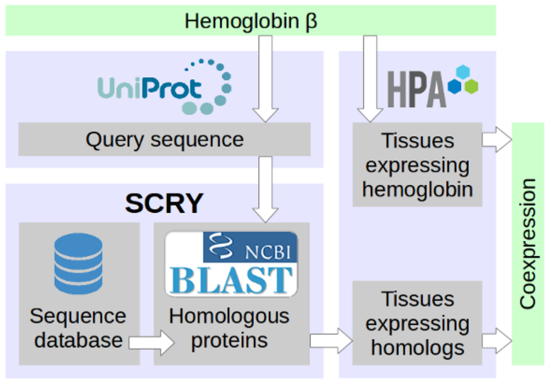
Use case 1: Statistics. We use the the Berlin SPARQL Benchmark (BSBM) [3] to evaluate our extension of SPARQL with generic statistical functions. This synthetic data set covers a set of products offered by different vendors, each given a rating by different reviewers. We want to know if any vendor's products are consistently rated higher than others, which we determine by calculating the mean, median and standard deviation of every product's rating, across all products sold by a specific vendor.

Note that calculating standard deviations requires computing square roots, which SPARQL does not natively support. Using SCRY, however, the query shown in figure 2 yields the answers by invoking two procedures defined in the `scry-math` package.

The graph pattern `{GRAPH ?g1 {math:median in: ?ratings; out: ?median.}}` instructs SCRY to 1) execute the procedure associated with the URI `math:median`, using whatever values are bound to `?ratings` as input; and 2) create an RDF (sub)graph matching the entire pattern, with the procedure's output inserted in the position of `?median`.

Use case 2: Bioinformatics. In this use case, we evaluate a very typical query from the life sciences. We use a protein sequence similarity search, to identify evolutionarily related (i.e. homologous) proteins. The derived relations are then used to determine which of a query protein's homologs are coexpressed in the same tissues.

Our `scry-blast` package defines two procedures: one which runs the most commonly used sequence similarity search method (BLAST [1]), given a protein sequence as input; and another which downloads such sequences from UniProt, given an identifier. Using hemoglobin β (P68871) as our protein of interest, we combine our procedures with



?tissue	?N
Alveolar macrophage	3
Bone marrow hematopoietic cell	4
Gall bladder glandular cell	3
Hemangioblast	4
Hepatic stellate cell	4
Hepatocyte	4
Lung macrophage	3
Pneumocyte	3

Fig. 3: Schematic representation of the bioinformatics use case query. Starting with hemoglobin β, have SCRY retrieve the sequence from UniProt. BLAST this sequence against a local database containing the sequences of every protein in the Human Protein Atlas, identifying proteins homologous to hemoglobin β. Query the Human Protein Atlas data for the tissues expressing hemoglobin β, and count how many homologs are coexpressed in those tissues. Results of running such a query are shown in the table on the right.

tissue-specific protein expression data from the Human Protein Atlas (HPA) [23]. As shown in figure 3, we can now write a query[6] which downloads hemoglobin β's sequence from UniProt, runs BLAST with that sequence to identify homologous proteins, and checks which tissues coexpress our protein of interest and its homologs.

Of course, SCRY can be extended with similar procedures to expose different similarity search methods, and queries can combine their results with other types of data.

Generality: SCRY enables users to integrate data processing procedures of arbitrary complexity in SPARQL queries, by embedding procedure calls in a query's graph patterns. This design is not restricted to procedures from or applications within any specific domain. Our use cases involved procedures which 1) integrate a service exposing simple mathematics and statistics functions to analyse BSBM data as shown in figure 2; and which 2) integrate a popular bioinformatics tool (BLAST) to derive evolutionary relationships between proteins at query time, as shown in figure 3.

These examples illustrate the simplicity and flexibility of using custom data processing methods within a SPARQL query, and that SCRY can extend queries with custom data processing methods of arbitrary complexity through intuitive, standards-compliant SPARQL syntax.

Reusability: The modular nature of services exposed through SCRY makes them easy to share, modify and reuse. For example, the standard deviation procedure used in the statistics use case can just as easily be applied to the e-values or molecular masses of proteins homologous to hemoglobin β identified in the bioinformatics use case. Once implemented, a SPARQL extension exposed through SCRY can be used in any query.

[6]See `http://tinyurl.com/scry4ls#file-coexpression`

Interoperability: The manner in which SCRY embeds procedure calls in query graph patterns is completely compatible with the SPARQL standard. Consequently, any standards-compliant SPARQL engine will accept queries that federate parts of the query to SCRY, e.g. with the `SERVICE` statement. To demonstrate, we loaded the benchmark data from the statistics use case into four popular triplestores: Apache Jena [20], OpenLink Virtuoso [18], OpenRDF Sesame [10] and Stardog [8]. We then succesfully resolved the query shown in figure 2 against each of these endpoints. SCRY is also compatible with query editors and similar tools, such as YASGUI [15].

Being able to resolve a single query against different endpoints, calling the same (reusable) procedure, sharply contrasts with the alternative: extending every triplestore individually, or using their built-in SPARQL extensions, and writing triplestore-specific queries for each. Making use of triplestore-specific built-in functions requires manual search through heterogeneous and cryptic documentation, if the desired built-in exists at all. Extending triplestores with custom functionality is similarly difficult: extending Sesame[7] with a simple function to calculate square roots requires 55 lines of Java[8] and deploying a properly documented JAR in a specific directory.

Note that in many cases, SPARQL queries can combine simple functions to express a more complex one. A standard deviation, for example, can be calculated by taking the square root of an array's variance, divided by its length. As complexity grows, though, such queries quickly become difficult to write and harder still to read[9]. In this context, SCRY provides an essential layer of abstraction, without which incorporating programs like BLAST would be difficult, tedious or outright impossible.

Scalability: As a SPARQL endpoint, SCRY has the same issues as every other; the time it takes to execute a query scales rather poorly with its complexity. Where service execution is concerned, however, SCRY offers two unique advantages over typical (Semantic) Web service platforms. First and foremost, SCRY is designed to be deployed with a personalized set of services and procedures, granting users full control over both the software and the hardware resolving their services. Secondly, it decouples the execution of those services from resolution of the query. This inherently makes services exposed by SCRY more scalable than equivalent web service implementations.

Running local implementations instead of addressing an equivalent web service can significantly improve the execution time of computationally expensive procedures. For example, running BLAST locally instead of through the NCBI's web service is several orders of magnitude faster (data not shown), in addition to giving users more control over critical parameters. Computationally expensive data processing methods are prominent in the life sciences, and using SCRY instead of online services to include their results in your queries can significantly speed up runtimes.

Note that resolving queries entirely locally has another significant advantage: privacy. Neither your query nor its (partial) results have to be exposed to external endpoints or (web) services to resolve, which is especially valuable when working with

[7]See `http://tinyurl.com/sesame-cookbook`

[8]See `http://tinyurl.com/sqrt-extensions#file-sesame-sqrt-java`

[9]See `http://tinyurl.com/sqrt-extensions` for triplestore-specific queries equivalent to that in figure 2

sensitive (e.g. clinical) data, from behind a firewall, or in an otherwise restricted network.

6 What does SCRY cost?

SCRY leverages SPARQL's query federation mechanism to make services general, reusable, interoperable and scalable, which comes at a cost. The SPARQL protocol requires communication between endpoints to occur through HTTP, which requires inputs and outputs for procedures exposed by SCRY to be serialized and deserialized to flat text. Additionally, SCRY needs time to parse embedded procedure calls, execute them, populate an RDF graph, and resolve a SPARQL query against it. For our statistics use case, which deals with a large number of inputs and outputs, these steps take up approximatetely 10% of the total query time.

In terms of reusability and standardization, SCRY provides optional description of the inputs and outputs of a procedure, but does not enforce their use. Consequently, it is not compatible with automated discovery of relevant services, unlike SADI for example, which registers services using formal semantics. This is a natural consequence of SCRY's focus on local computation, to enhance scalability, and our aim to be interoperable with extant triplestores and SPARQL tools.

7 Conclusion

We argue SCRY meets each of our requirements: **generality**, **reusability**, **interoperability** and **scalability**. Thus, computational biologists familiar with just the basics of SPARQL and Python can easily deploy SCRY, extend it with procedures of arbitrary complexity, and incorporate them in their SPARQL queries. SCRY overcomes issues common to the application of semantic web technology in the life sciences, such as the inability to incorporate computationally expensive data processing methods, or working with privacy sensitive data. It provides an essential layer of abstraction, without which incorporating programs like BLAST in SPARQL queries would be difficult, tedious or outright impossible.

Utilizing SCRY incurs some computational overhead, and the services it exposes are not standardized with sufficient rigidity to automate their discovery. Even so, we argue this is a price worth paying for a framework that offers such significant extension to SPARQL's functionality.

References

1. Altschul, S.F., et al.: Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Research (1997)
2. Battle, R., Kolas, D.: GeoSPARQL: Enabling a Geospatial Semantic Web. Semantic Web – Interoperability, Usability, Applicability (2011)
3. Bizer, C., Schultz, A.: The berlin sparql benchmark. International Journal on Semantic Web and Information Systems (2009)

4. Fielding, R.T.: Architectural styles and the design of network-based software architectures (2000)
5. Groth, P., et al.: API-centric Linked Data integration: The Open PHACTS Discovery Platform case study. Journal of Web Semantics 29(0), 12 – 18 (2014)
6. van Hage, W.R., with contributions from: Tomi Kauppinen, Graeler, B., Davis, C., Hoeksema, J., Ruttenberg, A., Bahls., D.: SPARQL: SPARQL client (2013), http://CRAN.R-project.org/package=SPARQL, R package version 1.15
7. Hoekstra, R., et al.: An Ecosystem for Linked Humanities Data. In: Proceedings of the Workshop on Humanities in the Semantic Web (WHiSe 2016), ESWC 2016 (2016)
8. Inc., C.: Stardog 3. Tech. rep., Complexible Inc. (2015), http://docs.stardog.com/
9. Krech, D., et al.: RDFLib Python Library. Tech. rep., RDFLib Team (2002), https://github.com/RDFLib/rdflib
10. OpenRDF: Openrdf sesame. http://rdf4j.org/
11. Pedrinaci, C., Domingue, J.: Toward the next wave of services: Linked Services for the Web of data. Journ. of Universal Computer Science 16(13), 1694—1719 (2010)
12. Queralt-Rosinach, N., Furlong, L.: DisGeNET RDF: a gene-disease association Linked Open Data resource . SWAT4LS (2013)
13. Redaschi1, N., the UniProt Consortium: UniProt in RDF: Tackling Data Integration and Distributed Annotation with the Semantic Web. Nature Precedings (2009)
14. Reynolds, D.: Linked Data API. Tech. rep., UK Government Linked Data (2009), https://github.com/UKGovLD/linked-data-api
15. Rietveld, L., Hoekstra, R.: The YASGUI Family of SPARQL Clients. Semantic Web – Interoperability, Usability, Applicability (2015)
16. Saleem, M., et al.: Linked Cancer Genome Atlas Database. Proceedings of the 9th International Conference on Semantic Systems (2013)
17. Schmachtenberg, M., et al.: The Semantic Web - ISWC 2014, chap. Adoption of the Linked Data Best Practices in Different Topical Domains, pp. 245–260 (2014)
18. Software, O.: Openlink virtuoso. http://virtuoso.openlinksw.com/
19. Speiser, S., Harth, A.: Integrating linked data and services with linked data services. In: The Semantic Web: Research and Applications. pp. 170—184. Springer (2011)
20. The Apache Software Foundation: Apache Jena. https://jena.apache.org/
21. The World Wide Web Consortium (W3C): Extending SPARQL Basic Graph Matching. https://www.w3.org/TR/rdf-sparql-query/#sparqlBGPExtend
22. The World Wide Web Consortium (W3C): SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/
23. Uhlé, M., et al.: Tissue-based map of the human proteome. Science (2015)
24. Vandervalk, B., McCarthy, L., Wilkinson, M.: SHARE & the Semantic Web - This Time it's Personal! Lecture Notes in Computer Science proceedings of the ASWC (2009)
25. W3C: SPARQL 1.1 Overview. https://www.w3.org/TR/sparql11-overview/
26. Wilkinson, M.D., et al.: The FAIR Guiding Principles for scientific data management and stewardship. Sci Data 3, 160018 (2016)
27. Wilkinson, M., et al.: The Semantic Automated Discovery and Integration (SADI) Web service Design-Pattern, API and Reference Implementation. Journal of Biomedical Semantics (2011)
28. Williams, A., et al.: Open PHACTS: semantic interoperability for drug discovery. Drug Discovery Today (2012)