Reflections on Profiling and Cataloguing the Content of SPARQL Endpoints using SPORTAL

Ali Hasnain[1] Syeda Sana e Zainab[1], Qaiser Mehmood[1], and Aidan Hogan[2]

[1] Insight Centre for Data Analytics, Galway, Data Science Institute (DSI) National University of Ireland (NUI), Galway

[2] Center for Web Research, Department of Computer Science, University of Chile

firstname.lastname@insight-centre.org

Abstract. This research focuses on the problem of helping clients find relevant SPARQL endpoints. To the best of author's knowledge, there are two online services that clients could use to try to find SPARQL endpoints based on their content: DATAHUB and/or VoID STORE. However, both services rely on static content descriptions provided by publishers themselves. Many of the endpoints listed in the DATAHUB have been offline for years; also, of the endpoints surveyed, VoID descriptions are only available in the DATAHUB for 33.3% and in the VoID STORE for 22.4%. Authors instead propose to compute extended VoID descriptions for public endpoints directly through their SPARQL interface. The SPORTAL service is available online at http://www.sportalproject.org/ originally introduced in journal publication[3]

1 Introduction

Rather than relying on publishers to compute and keep content descriptions up to date, we propose to compute content descriptions directly from the endpoints themselves. In particular, it is proposed to design a set of queries that can be issued to endpoints, where the results of these queries can then be used to build a catalogue that enables clients to find endpoints with relevant content with general descriptions. Such descriptions are normally not provided by the publishers of SPARQL endpoints independently of the endpoint itself. This work explores the feasibility of computing content descriptions directly from SPARQL endpoints. More concretely, SPORTAL (SPARQL PORTAL) is proposed which is a centralised catalogue indexing content descriptions computed from individual SPARQL endpoints [4,3]. The goal of SPORTAL is to help both human and software agents to find public SPARQL endpoints relevant for their needs. SPORTAL relies only on SPARQL queries to gather information about the content of each endpoint and hence only assumes a working SPARQL interface rather than requiring the publishers hosting endpoints to provide additional descriptions of the datasets. Queries were issued to each endpoint to gather metadata about its content, which are later used to find relevant endpoints. Taking a simple example, instead of querying each endpoint every time an agent is looking for a given class, can occasionally query each endpoint for an up-to-date list of their classes and use that list to find relevant endpoints for the agent at

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2576/paper03.pdf

runtime. One of the main design questions for SPORTAL then is: what content descriptions should such a system try to compute from endpoints? Ideally the content descriptions should be as general as possible, supporting a variety of different types of clients and searches.With the advent of novel features in SPARQL 1.1 like aggregates, it is possible to formulate queries that ask, e.g., *how many triples the knowledge base contains, which classes or properties are used*, *how many unique instances of each class appears*, *which properties are used most frequently with instances of which classes* etc. SPARQL endpoints can be considered self-descriptive if they can describe their own content[5]. SPORTAL is limited due to practical thresholds on the amount of data that a SPARQL endpoint will return. Given that many endpoints contain millions of resources and text literals, this rules out, for example, building a complete inverted index over the content of an individual endpoint, or indexing all resources that an endpoint mentions. SPORTAL compute concise content descriptions rather than mirroring remote endpoint content. Thus, the research focuses on computing concise, schema-level descriptions of endpoints. Using such descriptions, one can directly find relevant endpoints and can indirectly help with other forms of queries (e.g., *to find endpoints that contain instances of GENE, though they may not necessary be from a rat*). Extended Vocabulary of Interlinked Datasets (VoID) descriptions [7] was computed as VoID is also used in federated scenarios to find relevant endpoints [1,2,6] . Results shows while 93.8% of operational endpoints respond successfully when asked for *list of classes*, only 40.2% respond successfully when additionally asked how many instances those classes have. Thus, the SPORTAL catalogue would include metadata about the classes that appear in 93.8% of the catalogued endpoints, but only in 40.2% cases would the catalogue have information about how many instances appear in those classes.

2 Self- Descriptive Queries

This section lists the set of SPARQL 1.1 queries that are used to compute a VoID-like description from the content indexed by an endpoint.

2.1 Functionality

We test the availability and SPARQL 1.1 compliance of an endpoint using two queries with features: sub-queries and the count aggregate function.

2.2 Dataset-level statistics

A set of "dataset-level" queries is listed in Table 1 that form a core part of VoID to ascertain the number of triples (Q_{B1}), and the number of distinct classes (Q_{B2}), properties (Q_{B3}), subjects (Q_{B4}), and objects (Q_{B5}). These queries require support for SPARQL 1.1 COUNT and sub-query features. Using these agents can use them to find endpoints indexing datasets that fall within a given range of triples in terms of overall size, or, for example, to find the endpoints with the largest datasets. Counts may be particularly useful - in combination with later categories - to order the endpoints; for example, to find the endpoints with a given class and order them by the total number of triples they index.

Table 1: Queries for dataset-level VoID statistics

No	Query
Q_{B1}	CONSTRUCT <D> v:triples ?x WHERE SELECT (COUNT(*) AS ?x) WHERE ?s ?p ?o
Q_{B2}	CONSTRUCT <D> v:classes ?x WHERE SELECT (COUNT(DISTINCT ?o) AS ?x) WHERE ?s a ?o
Q_{B3}	CONSTRUCT <D> v:properties ?x WHERE SELECT (COUNT(DISTINCT ?p) AS ?x) WHERE ?s ?p ?o
Q_{B4}	CONSTRUCT <D> v:distinctSubjects ?x WHERE SELECT (COUNT(DISTINCT ?s) AS ?x) WHERE ?s ?p ?o
Q_{B5}	CONSTRUCT <D> v:distinctObjects ?x WHERE SELECT (COUNT(DISTINCT ?o) AS ?x) WHERE ?s ?p ?o

3 Class-based statistics

Third similar statistics about the instances of each class following the notion of class partitions in VoID were ascertained: a subset of the data considering only triples where instances of that class are in the subject position. Table 2 lists the six queries used. The first query Q_{C1} merely lists all class partitions. The other five queries ($Q_{C2\text{–}6}$) count the triples and distinct classes, predicates, subjects and objects for each class partition; e.g., Q_{C2} retrieves the number of triples where instances of that class are in the subject position. Queries $Q_{C2\text{–}6}$ introduce COUNT, sub-queries and also GROUPBY features from SPARQL 1.1.

Table 2: Queries for statistics about classes

No	Query
Q_{C1}	CONSTRUCT <D> v:classPartition [v:class ?c] WHERE ?s a ?c
Q_{C2}	CONSTRUCT v:classPartition [v:class ?c ; v:triples ?x] WHERE SELECT (COUNT(?p) AS ?x) ?c WHERE ?s a ?c ; ?p ?o GROUP BY ?c
Q_{C3}	CONSTRUCT v:classPartition [v:class ?c ; v:classes ?x] WHERE SELECT (COUNT(DISTINCT ?d) AS ?x) ?c WHERE ?s a ?c , ?d GROUP BY ?c
Q_{C4}	CONSTRUCT v:classPartition [v:class ?c ; v:properties ?x] WHERE SELECT (COUNT(DISTINCT ?p) AS ?x) ?c WHERE ?s a ?c ; ?p ?o GROUP BY ?c
Q_{C5}	CONSTRUCT v:classPartition [v:class ?c ; v:distinctSubjects ?x] WHERE SELECT (COUNT(DISTINCT ?s) AS ?x) ?c WHERE ?s a ?c GROUP BY ?c
Q_{C6}	CONSTRUCT v:classPartition [v:class ?c ; v:distinctObjects ?x] WHERE SELECT (COUNT(DISTINCT ?o) AS ?x) ?c WHERE ?s a ?c ; ?p ?o GROUP BY ?c

Once catalogued, agents can use statistics describing class partitions of the datasets to find endpoints mentioning a given class, where they can additionally (for example) sort results in descending order according to the number of unique instances of that class, or triples used to define such instances, and so forth. Hence the counts computed by ($Q_{C2\text{–}6}$) help agents to distinguish endpoints that may only have one or two instances of a class to those with thousands or millions. Likewise, criteria can be combined arbitrarily for multiple classes, or with the overall statistics computed previously.

3.1 Property-based statistics

This section looks at property partitions in the dataset, where a property partition refers to the set of triples with that property term in the predicate position. Queries are listed in Table 3. As before, Q_{D1} lists the property partitions. $Q_{D2\text{-}4}$, count the number of triples, distinct subjects and distinct objects. Using these statistics about property partitions in the catalogue, agents can, for example, retrieve a list of public endpoints using a given property ordered by the number of triples using that specific property. Likewise criteria can be combined arbitrarily for multiple properties, or with the dataset- or class-level metadata previously collected; for example, an agent may wish to order endpoints by the ratio of triples using a given property (where the count from Q_{D2} for the property in question can be divided by the total triple count from Q_{B1}), or to find endpoints where all subjects have an rdfs:label value (where the count computed from Q_{D3} for that property should match the count for Q_{B4}).

Table 3: Queries for statistics about properties

No	Query
Q_{D1}	`CONSTRUCT v:propertyPartition [ v:property ?p ] WHERE ?s ?p ?o`
Q_{D2}	`CONSTRUCT v:propertyPartition [ v:property ?p ; v:triples ?x ]` `WHERE SELECT (COUNT(?o) AS ?x) ?p WHERE ?s ?p ?o GROUP BY ?p`
Q_{D3}	`CONSTRUCT v:propertyPartition [ v:property ?p ; v:distinctSubjects ?x ]` `WHERE SELECT (COUNT(DISTINCT ?s) AS ?x) ?p WHERE ?s ?p ?o GROUP BY ?p`
Q_{D4}	`CONSTRUCT v:propertyPartition [ v:property ?p ; v:distinctObjects ?x ]` `WHERE SELECT (COUNT(DISTINCT ?o) AS ?x) ?p WHERE ?s ?p ?o GROUP BY ?p`

3.2 Nested class–property statistics

Fifth, the focus is to look at how classes and properties are used together in a dataset, gathering statistics on property partitions nested within class partitions: these statistics detail how properties are used for instances of specific classes. Table 4 lists the four queries used. Q_{E1} lists the property partitions nested inside the class partitions, and Q_{E2-4} count the number of triples using a given predicate for instances of that class, as well as the number of distinct subjects and objects those triples have.An agent could use the resulting metadata to find endpoints describing instances of specific classes with specific properties, with filtering or sorting criteria based on, e.g., the number of triples. For example, an agent might be specifically interested in images of people, where they would be looking for the class-partition foaf:Person with the nested property-partition foaf:depicts. It would have been possible to find endpoints that have data for the class foaf:Person and triples with the property foaf:depicts, but not that the images were defined for people.

3.3 Miscellaneous statistics

In the final set of experiments, the focus is to look at queries that yield statistics not supported by VoID as listed in Table 5. In particular, experiments were designed to see if endpoints can return a subset of statistics from the VoID Extension Vocabulary , which include counts of different types of unique RDF

Table 4: Queries for nested property/class statistics

No	Query
Q_{E1}	`CONSTRUCT { v:classPartition [ v:class ?c ; v:propertyPartition [ v:property ?p } ] ] WHERE { ?s a ?c ; ?p ?o }`
Q_{E2}	`CONSTRUCT { v:classPartition [ v:class ?c v:propertyPartition [ v:property ?p ;} v:triples ?x ] ] WHERE { SELECT (COUNT(?o) AS ?x) ?p WHERE { ?s a ?c ; ?p ?o } GROUP BY ?c ?p }`
Q_{E3}	`CONSTRUCT { v:classPartition [ v:class ?c ; v:propertyPartition [ v:distinctSubjects ?x ] ] WHERE { SELECT (COUNT(DISTINCT ?s) AS ?x) ?c ?p} WHERE { ?s a ?c ; ?p ?o } GROUP BY ?c ?p }`
Q_{E4}	`CONSTRUCT { v:classPartition [ v:class ?c ; v:propertyPartition [ v:distinctObjects ?x ;} v:property ?p ] ] WHERE { SELECT (COUNT(DISTINCT ?o) AS ?x) ?c ?p WHERE { ?s a ?c ; ?p ?o }} GROUP BY ?c ?p`

terms in different positions: subjects IRIs (Q_{F1}), subject blank nodes (Q_{F2}), objects IRIs (Q_{F3}), literals (Q_{F4}), object blank nodes (Q_{F5}), all blank nodes (Q_{F6}), all IRIs (Q_{F7}), and all terms (QF8). Inspired by the notion of "schema maps" as proposed by Kinsella et al. [8], queries also count the classes that the subjects and objects of specific properties are instances of (Q_{F9-10}); these are "inverses" of queries (Q_{E3-4}). Using the resulting data the agent in question could look for datasets without any blank nodes or for datasets where a given number of the objects of a given property are of a certain type. Likewise, the user can find endpoints with more than ten million triples where at least 30% of the unique object terms are literals.

4 Experiments and Results

This section looks at how public SPARQL endpoints themselves perform for the list of previously enumerated self-descriptive queries. A list of 540 SPARQL endpoints registered in the DATAHUB along with a list of 137 endpoints from Bio2RDF releases 1-3 was collected resulting in total of 618 unique endpoints (59 endpoints were present in both lists) were considered.

4.1 Implementation Used

Although there is no generic exact method of determining the engine powering a SPARQL endpoint, the HTTP header may contain some clues in the Server field. Hence the first step was to perform a lookup on the endpoint URLs. The response codes of this step, where quite a large number of endpoints return error codes 4xx, 5xx, or some other exception. This indicates that a non-trivial fraction of the endpoints from the list are offline. With respect to the server names returned by those URLs that returned a HTTP response, although some of the server names denote generic HTTP servers - more specifically Apache, nginx, Jetty, GlassFish, Restlet and lighttpd - some names that indicate SPARQL implementations - namely Virtuoso, Fuseki and 4s-httpd (4store).

4.2 Availability and version

Hence, the next step was to look at how many endpoints respond to the basic availability query Q_{A1}. Given that the queries were run in an uncontrolled en-

Table 5: Queries for miscellaneous statistics

No	Query
Q_{F1}	`CONSTRUCT e:distinctIRIReferenceSubjects ?x WHERE SELECT (COUNT(DISTINCT ?s ) AS ?x) WHERE ?s ?p ?o FILTER(isIri(?s))`
Q_{F2}	`CONSTRUCT e:distinctBlankNodeSubjects ?x WHERE SELECT (COUNT(DISTINCT ?s) AS ?x) WHERE ?s ?p ?o FILTER(isBlank(?s))`
Q_{F3}	`CONSTRUCT e:distinctIRIReferenceObjects ?x WHERE SELECT (COUNT(DISTINCT ?o ) AS ?x) WHERE ?s ?p ?o FILTER(isIri(?o))`
Q_{F4}	`CONSTRUCT e:distinctLiterals ?x WHERE SELECT (COUNT(DISTINCT ?o ) AS ?x) WHERE ?s ?p ?o FILTER(isLiteral(?o))`
Q_{F5}	`CONSTRUCT e:distinctBlankNodeObjects ?x WHERE SELECT (COUNT(DISTINCT ?o ) AS ?x) WHERE ?s ?p ?o FILTER(isBlank(?o))`
Q_{F6}	`CONSTRUCT e:distinctBlankNodes ?x WHERE SELECT (COUNT(DISTINCT ?b ) AS ?x) WHERE ?s ?p ?b UNION ?b ?p ?o FILTER(isBlank(?b))`
Q_{F7}	`CONSTRUCT e:distinctIRIReferences ?x WHERE SELECT (COUNT(DISTINCT ?u ) AS ?x) WHERE ?u ?p ?o UNION ?s ?u ?o UNION ?s ?p ?u FILTER(isIri(?u)`
Q_{F8}	`CONSTRUCT e:distinctRDFNodes ?x WHERE SELECT (COUNT(DISTINCT ?n ) AS ?x) WHERE ?n ?p ?o UNION ?s ?n ?o UNION ?s ?p ?n`
Q_{F9}	`CONSTRUCT v:propertyPartition [ v:property ?p ; s:subjectTypes [ s:subjectClass ?sType ; s:distinctMembers ?x ] ] WHERE SELECT (COUNT(?s) AS ?x) ?p ?sType WHERE ?s ?p ?o ; a ?sType . GROUP BY ?p ?sType`
Q_{F10}	`CONSTRUCT v:propertyPartition [ v:property ?p ; s:objectTypes [ s:objectClass ?oType ; s:distinctMembers ?x ] ] WHERE SELECT (COUNT(?o) AS ?x) ?p ?oType WHERE ?s ?p ?o . ?o a ?oType . GROUP BY ?p ?oType`

vironment, multiple runs were performed to help mitigate temporary errors and remote server loads: the core idea is that if an endpoint fails at a given moment of time, a catalogue could simply reuse the most recent successful result. Along these lines, three weekly experiments were run. In total, 306 endpoints (49.5%) responded to Q_{A1} at least once in the three weeks; these endpoints were considered to be operational and others to be offline. Of the operational endpoints, 7 (1.1%) responded successfully exactly once to Q_{A1}, 28 (4.5%) responded successfully exactly twice, and 272 (44.1%) responded successfully thrice. In the most recent run, 298 endpoints responded to Q_{A1}. Of these, 168 (56.4%) also responded with a single result for Q_{A2}, indicating some support for SPARQL 1.1 in about half of the operational endpoints.

4.3 Success rates

The focus is on the overall success rates for each query, looking at the ratio of the 307 endpoints that return non-empty results. Figure 1, shows the success rates varying from 25% for Q_{E3} on the lower end, to 94% for Q_{C1} on the higher end. The three queries with the highest success rates require only SPARQL 1.0 features to run: list all class partitions (Q_{C1}), all property partitions (Q_{D1}), and all nested partitions (Q_{E1}). Hence, only 49% could respond to the SPARQL 1.1 test query Q_{A1} - more endpoints can answer queries not requiring novel SPARQL 1.1 features such as counts or sub-queries. The query with the highest success rate

that involved SPARQL 1.1 features was Q_{B1}, where 51% of endpoints responded with a count of triples. In general, queries deriving counts within partitions had the lowest success rates

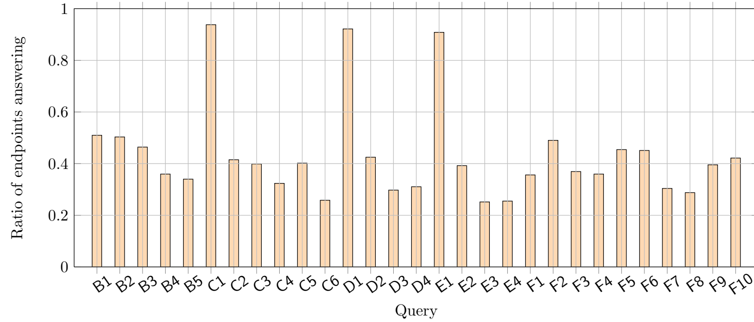

Fig. 1: Ratio of endpoints returning non-empty results per query

4.4 Result Size

Figure 2 shows result sizes in log scale for individual queries at various percentiles considering all endpoints that returned a non-empty result. As expected, queries that return a single count triple return one result across all percentiles. For other queries, the result sizes extended into the tens of thousands. One may note that the higher percentiles are quite compressed for certain queries, indicating the presence of result thresholds. For example, for Q_{C1}, a common result-size was precisely 40,000, which would appear to be the effect of a result-size threshold. Hence, unlike the local experiments where result thresholds could be switched off, for public endpoints, partial results are sometimes returned.

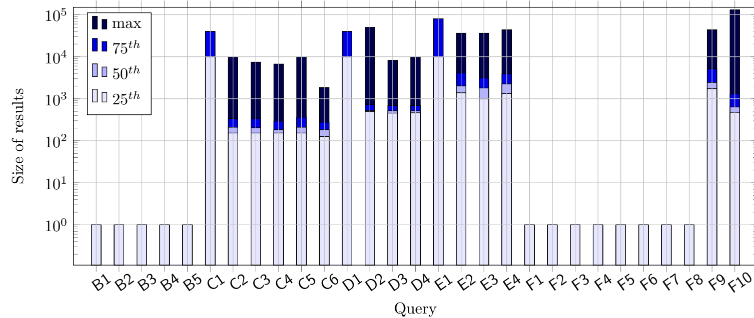

Fig. 2: Sizes of results for different queries taking 25th, 50th (median), 75th and 100th (max) percentiles, inclusive, across endpoints for non-empty results

4.5 Runtimes

Finally, the focus was on runtimes for successfully executed queries, incorporating the total response time for issuing the query and streaming all results. Figure

3, presents the runtimes for each query considering different percentiles across all endpoints returning non-empty results in log scale. A large variance in runtimes is noticed, which is to be expected given that endpoints host datasets of a variety of sizes and schemata on servers with a variety of computational capacity. In general, the 25th percentile roughly corresponds with the one second line, but that slower endpoints may take tens or hundreds of seconds. The at max trend seems to be the effect of remote timeout policies, where query runtimes often maxed out at between 100 - 120 seconds, likely returning partial results.

5 SPARQL Portal

Our primary motivation is to investigate a method for cataloguing the content of public SPARQL endpoints without requiring them to publish separate, static descriptions of their content-or indeed, for publishers to offer any additional infrastructure other than the query interface itself. This section, describes the SPORTAL catalogue itself, including its interfaces, capabilities and limitations.

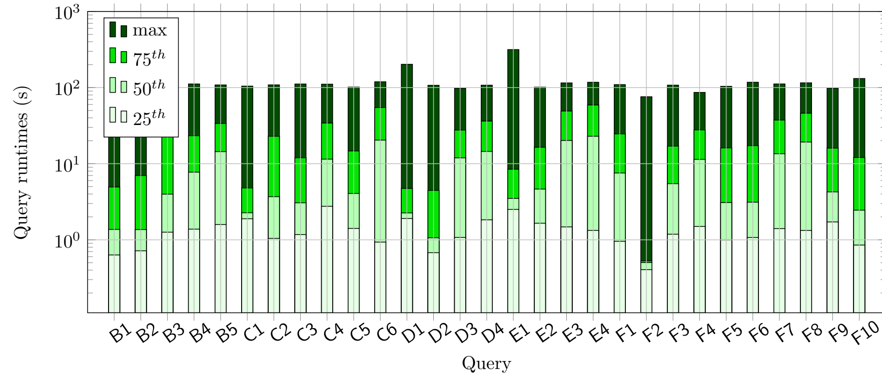

Fig. 3: Runtimes for different queries taking 25th, 50th (median), 75th and 100th (max) percentiles inclusive, across all endpoints returning non-empty results

5.1 SPARQL interface

SPORTAL, itself provides a public SPARQL endpoint, where the RDF triples produced by the CONSTRUCT clauses of the self-descriptive queries issued against public endpoints can themselves be queried. This allows users with specific requirements in mind to interrogate the catalogue in a flexible manner. Let us consider *an agent looking for SPARQL endpoints with at least 50 unique images of people*, where this agent may ask the following query with results:

Query for 50 images

Query
SELECT DISTINCT ?endpoint ?imgs
WHERE { ?dataset v:classPartition
[v:class f:Person ; v:propertyPartition [
v:property f:depiction; v:distinctObjects ?imgs]] ;
v:sparqlEndpoint ?endpoint . FILTER(?imgs ¿ 50) }
ORDER BY DESC (?imgs)

Results with 50 images

?endpoint	?imgs
http://eu.dbpedia.org/sparql	4,517
http://eudbpedia.deusto.es/sparql	4,517
http://data.open.ac.uk/query	311

5.2 User Interface

To help non-expect users, SPORTAL also provides an online user interface with a number of functionalities. First, users can search for specific endpoints by their URL, by the classes in their datasets, and/or by the properties in their datasets. These features are offered by means of auto-completion on keywords, meaning that the agent need not know the specific IRIs they are searching for. If a user clicks on or searches for an endpoint, they can retrieve all the information available about that endpoint as extracted by the queries previously described, providing an overview of how many triples it contains, how many subjects, how many classes, etc. (as available). The SPORTAL user interface also includes some graphical visualisations of some of the high-level features of the catalogue, such as the most popular classes and properties based on the number of endpoints in which they are found, the most common server headers, and so forth. While this may not be of use to a user with a specific search in mind, it offers a useful overview of the content available across all endpoints on the Web, and the schema-level terms that are most often instantiated.

References

1. Hasnain, A., Kamdar, M.R., Hasapis, P., Zeginis, D., Warren Jr, C.N., et al.: Linked Biomedical Dataspace: Lessons Learned integrating Data for Drug Discovery. In: International Semantic Web Conference (In-Use Track), October 2014 (2014)
2. Hasnain, A., Mehmood, Q., Sana e Zainab, S., Saleem, M., Warren, C., Zehra, D., Decker, S., Rebholz-Schuhmann, D.: Biofed: federated query processing over life sciences linked open data. Journal of Biomedical Semantics 8(1), 13 (2017), http://dx.doi.org/10.1186/s13326-017-0118-0
3. Hasnain, A., Mehmood, Q., e Zainab, S.S., Hogan, A.: Sportal: Profiling the content of public sparql endpoints. International Journal on Semantic Web and Information Systems (IJSWIS) 12(3), 134–163 (2016), http://www.igi-global.com/article/sportal/160175
4. Hasnain, A., Mehmood, Q., e Zainab, S.S., Hogan, A.: SPORTAL: searching for public SPARQL endpoints. In: Proceedings of the ISWC 2016 Posters & Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016), Kobe, Japan, October 19, 2016. (2016), http://ceur-ws.org/Vol-1690/paper78.pdf
5. Hasnain, A., Mehmood, Q., e Zainab, S.S., Hogan, A.: Cataloguing the context of public sparql endpoints. In: Innovations, Developments, and Applications of Semantic Web and Information Systems, pp. 295–328. IGI Global (2018)
6. Hasnain, A., e Zainab, S.S., Zehra, D., Mehmood, Q., Saleem, M., Rebholz-Schuhmann, D.: Federated query formulation and processing through biofed. In: SeWeBMeDA@ ESWC. pp. 16–19 (2017)
7. Hausenblas, M., Cyganiak, R.: Describing linked datasets-on the design and usage of void, the'vocabulary of interlinked datasets' (2009)
8. Kinsella, S., Bojars, U., Harth, A., Breslin, J.G., Decker, S.: An interactive map of semantic web ontology usage. In: 2008 12th International Conference Information Visualisation. pp. 179–184. IEEE (2008)