Hybrid Federations

Connecting Web APIs and Linked Data Knowledge Bases

Tobias Zeimetz Trier University
54286 Trier, Germany
zeimetz@uni-trier.de

ABSTRACT

The research plan described in this article is intended to develope a system which can help data curators, data scientists, and other users in the domain of Linked Data to identify important data sources, understand their structure, and their schema. In addition, the system should be easy to use for non-expert users so that they can quickly and easily formulate more complex queries, e.g by using a visual interface. Furthermore, Linked Data Federations will be extended to include Web APIs as knowledge bases, denoted as Hybrid Federations. By using Web APIs it should be made possible to integrate so-called user defined functions (e.g. similarity search) into SPARQL.

Keywords

Record Linkage, Schema Inference, Hybrid Federations

1. INTRODUCTION

The possibility to link different sorts of knowledge bases (e.g., dblp [3], WikiData [10] or DBpedia [4]) is one of the main strengths of Linked Open Data. Also, the usage of different ontologies (e.g., FOAF [5]) to give semantics (i.e. meaning) to the data is a great advantage. However, Linked Open Data also has drawbacks that go along with the advantages. The wide selection of ontologies can tempt to define own properties or predicates because developers first need to understand the structure of the various ontologies. Especially if an ontology is not as granular as the used data structure (i.e. if the ontology is very detailed, but data is rather high-level or the other way around), developers often tend to create their own properties. For these reasons it can sometimes be a hard task to get an overview of a new knowledge base.

With relational databases, a user can display the schema to get an overview of the data set. However, since Linked Open Data is a graph database there is no schema needed. Linked Data is stored in RDF [7] format, which is a standard model for data interchange on the Web. In order to discover the schema of a (RDF) database, a user has to formulate multiple queries. The query language for RDF is called SPARQL [8]. A SPARQL query consists of triple patterns, conjunctions, disjunctions and so on. The triples are composed of subject (start node), property (directed edge) and an object (target node).

Several systems [19, 12, 11, 13, 14, 15, 6, 20, 22] have been developed to help extract the schema from a knowledge base and graphically display it to a non-expert user. Most approaches are so-called offline approaches, where the user needs to download a data dump and extract the schema of the downloaded RDF files offline. Such approaches have some disadvantages, such as that the provided data dumps are not up-to-date or that not every data provider provides downloadable data dumps.

Only few systems extract the schema using the SPARQL Endpoint of the knowledge base. This approach has the benefits that we do not need to process data dumps and that the information is as up-to-date as possible. However, such approaches have disadvantages that we need to overcome. Typical problems for example are the response time of the SPARQL endpoints or the fact that sometimes no response (depending on the complexity of the query) is delivered at all. For this reason a goal of our research is to overcome these limitations and find a way to extract the schema even for big knowledge bases such as WikiData [10] or DBpedia [4].

Furthermore, we connect Linked Open Data in form of SPARQL endpoints with Web APIs, e.g. CrossRef [2] or Springer SciGraph [9]. By connecting knowledge bases and Web APIs it is possible to create a so called *Hybrid Federation*. A federation is a combination of several knowledge bases, which can then be queried like a homogeneous system.

As described in [24, 23] knowledge base management use cases often require addressing hybrid information needs that involve multiple different data sources, data modalities (e.g. similarity, topic or keyword search) and the availability of computation services (e.g. graph analytics algorithms). In SPARQL however, the support for hybrid information needs is very limited. Therefore, we extend the SPARQL query language by user defined functions, e.g. keyword or similarity search. To realize this step, we use again Web APIs, so that a user can develop a (local) Web API and then embed it in SPARQL as a service. By calling this service, the function implemented by the Web API is to be processed in the SPARQL Query Language.

The remaining part of the article is structured as follows: Section 2 shows some use cases in which a hybrid federation or the visualization of a schema can be helpful. Afterwards

31^{st} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 11.06.2019 - 14.06.2019, Saarburg, Germany.

we present our research plan in Section 4. There we explain which problems need to be solved and go deeper into the details of Hybrid Federations. In Section 5 we present our evaluation plan and some data sets we want to use. The last section gives a brief overview of related work such as LODeX [12, 11, 6, 13, 14, 15] or FacetGraphs [18].

2. USE CASES

In this part we present several use cases in detail. The first use case tackles the problem of non-experts or non-tech users, i.e. it should be possible for a user (without knowledge about SPARQL) to understand the structure of a knowledge base in a fast way. Furthermore, the extraction of a schema should make it possible to identify important (new) knowledge bases. In addition, it should be easy for such a user to connect data from a knowledge base with the data of a Web API. For this reason we motivate the use of a visual query interface, like presented by FacetGraphs [18] or LODeX [13, 14].

The second use case deals with the integration of data. In this case, information from a Web API (e.g. Springer SciGraph [9] or CrossRef [2]) is to be added to an existing knowledge base (e.g. dblp[3]). The enrichment of (especially) meta data for publications and authors is very interesting from a data curator's point of view, because it can be used to provide more data about publications to the users, to disambiguate authors and to find erroneous data in used knowledge bases.

The last use case, namely the data processing use case, tackles the problem of domain oriented functions, e.g. similarity search, topic analysis and more. For example, until today it is not possible to execute a SPARQL query like "Give me all articles that are similar to the article with the DOI *d*". SPARQL does not understand the concept of similarity and therefore a user has to implement a program to solve this task. This use case illustrates the need to implement user defined functions within SPARQL.

2.1 Non-Expert Use Case

A common use case for data curators is finding new and relevant data sets. The curator has to look at the data in the new data set and find out whether these data fit to his database (federation) at all. If, for example, the underlying database is of bibliographic nature, it makes little sense to search for more information in a sport's database. A curator therefore needs to be able to quickly determine the domain of a data set (SPARQL endpoint).

Unlike relational databases, graph databases like Linked Data knowledge bases do not necessarily require a schema. A curator has to formulate multiple queries to discover the structure of the schema and knowledge base. Depending on the complexity and size of the explored knowledge base, this can require several complex SPARQL queries and can be a time-consuming task. Furthermore, the curator needs to know how to formulate SPARQL queries (expert user).

Most data curators, are non-expert users and even if they were experts, it would still take some time to figure out the structure and domain of the database.For this reason, this use case focuses on extracting and visualizing the structures of a Linked Data knowledge base. Further, we consider the problem that non-expert users may still have to formulate their own queries in order to obtain detailed information. However, since the concepts of a query language must be understood by the curator, an alternative query method is needed in this use case. Here the curator should have the possibility to formulate complex queries with a few clicks via a graphical user interface.

2.2 Data Integration Use Case

The dblp computer science bibliography [3] is a collection of bibliographic meta data on major computer science publications. To extend and improve the information stored in dblp it is important to collect data from different data repositories such as Springer SciGraph [9], CrossRef [2] and more. The new gained (meta) data can be used for several tasks such as identifying erroneous data in current knowledge bases or to disambiguate authors.

The usual process is to download (not up-to-date) data dumps and to integrate the downloaded data into the dblp data repository by using self coded scripts or programs. The main problems in this approach are (1) that the used data is not up-to-date and (2) that data providers often change the structure of the data dumps (new tags, different structures, etc.) such that the used programs and crawler needs to be changed.

Especially the last task is very bothersome, because it is not uncommon that programs and crawler have to be changed completely in order to work again correctly. For this reason it is desirable to query a Linked Data repository and combine it with the data provided via other endpoints e.g. Web APIs. Because also the schema of endpoints can change it is important that the algorithms, to combine the data of endpoints with APIs, can automatically detect linkage points. Furthermore, the user should not notice that some data providers do not provide a SPARQL endpoints. The goal is, that the user has the feeling of a homogeneous database while querying but in reality using different data formats, data modalities and different kinds of endpoints (denoted as hybrid federation). It should also be possible to extend the used data sources quickly and use different kinds of endpoints such as SciGraph [9] or CrossRef [2].

2.3 Data Processing Use Case

In the previous use case we wanted to integrate data into an existing knowledge base by using multiple heterogeneous data repositories and formats (called a hybrid federation). The next step is to work with this information and process the data, e.g. by using data mining or data analysis techniques. One example is a query that filters all publications similar to a previously specified publication: "Select all publications that are similar to the publication with DOI *d*".

SPARQL provides some basic functions such as filter the minimum, maximum or a count function. But more advanced and domain oriented tasks like a similarity search based on abstracts are not included in the SPARQL Query Language. For this reason it is desirable to add user defined functions to the toolbox of SPARQL which can be defined/implemented by a developer (expert user) and in addition, can be fast and easy adopted into SPARQL.

3. RELATED WORK

Some work has already been done in the area of Schema Inference. Also, the user defined functions were already introduced in [24, 23]. In the following we take a closer look at the previous work.

3.1 Schema Inference

As already mentioned, Schema Inference can be divided into two groups. The first group of algorithms works on data dumps and can therefore ignore server problems (offline approach). However, the problem with this approach is that the data dumps are usually not up-to-date. The second group tries to extract the schema via the SPARQL endpoints (online approach) and can therefore work on current data.

SchemEx [20] is a system that processes data dumps and extracts the schema from them. However, this approach is not able to retrieve the properties among classes because it does not consider class instances.

In contrast, LODeX [12, 11, 6, 15] proposes an approach that creates a set of indexes that enhance the description of the knowledge base. As Benedetti et al. state, these indexes collect statistical information regarding the size and complexity of the knowledge base (e.g. number of instances), but also present all the instantiated classes and the properties among them. The main problem in the approach of LODeX is that it does not work on large endpoints such as Wikidata [10] or DBpedia [4].

Since not all classes of an endpoint are needed in order to determine the domain of the knowledge base, LD-VOWL [22] extracts only the top k classes. It provides in addition a visualization of the top k classes and properties in a knowledge base. A big advantage of this approach is that only the most used schema information is extracted and the user is not flooded with information. The major disadvantage of this approach is that operators like ORDER BY must be used. Especially weak servers or servers with large amounts of data are quickly brought to their limits.

Kellou-Menouer and Kedad present in SchemaDecrypt [19] an approach, for discovering a versioned schema for SPARQL Endpoints. SchemaDecrypt enables the discovery of the different structures of the existing classes in a knowledge base. This is an interesting approach because it shows which versions of classes and types exist. These are characterized above all by the fact that a version of a class is created by combining different properties.

Approaches as described in [12, 11, 6, 15, 19, 22] still suffer in scalability or soundness, e.g. it is not possible to extract the schema of large knowledge bases or the extracted schema is missing important connections and/or adds additional connections that do not exist.

3.2 (Hybrid) Federations

The idea of federations has been around for a long time and several systems like FedX [27, 26], SPLENDID [17] or SCRY [28] have been developed. All these systems only work on SPARQL endpoints and do not integrate other data sources such as Web APIs. The step to hybrid federations is introduced by Koutraki et al. in [21]. They present a system that combines data across different Web APIs and can automatically infer the view definition in a global schema. Koutraki et al. state that the system can automatically infer the schema with precision of 81%-100%. However, the problem of manually configuring the input types of a Web API still remains.

Preda et al. introduced with ANGIE [25] a system that can answer queries by combining local knowledge bases and Web APIs. If a query cannot be answered by querying the used knowledge base, the system calls the corresponding Web API in order to retrieve the missing information. The presented system is a hybrid federation of various data sources where some information is stored locally and other is mapped into the local knowledge base on demand. Preda et al. call this approach an *active knowledge base*.

Web APIs are viewed as functions in ANGIE which are modeled as an RDF graph that contains variables like a query. This means that, similiar as in the System of Koutraki et al., a user needs to configure the Web API manually.

3.3 User Defined Functions

Most approaches that are focused on hybrid query processing share the assumption that federation members provide their data in Linked Data formats such as RDF. Domain oriented functions such as similarity search are supported by using special indices and predefined properties (e.g., full-text search in Virtuoso). This is not a general/sufficient solution, because not every knowledge base provides these indices.

Some work in this domain is done by Nikolov et al. introducing the Ephedra system [24, 23]. It is a SPARQL federation engine that provides the possibility to process hybrid queries using the SERVICE and BIND keywords. With this approach Nikolov et al. make it possible to connect SPARQL endpoints and RESTful web services. Furthermore, it provides a mechanism to include hybrid services into SPARQL federations. In addition, they implement various query optimization techniques, thereby the focus is on two types of improvements: Join order optimization and assigning appropriate executors for JOIN and UNION operators.

4. RESEARCH PLAN

In the following section we describe in more detail what we want to implement and what we want to improve. First, our research aims to develop a scalable, efficient and sound algorithm to derive the schema of a SPARQL endpoint. We then present a basic procedure for linking the data from Web APIs to the data of SPARQL endpoints. Furthermore, we implement (as an intermediate) step a graphical query interface, so that non-expert user can formulate complex queries. In addition, we present a basic idea, how to combine Web APIs and SPARQL service calls to realize user defined functions.

4.1 Online Schema Inference

This section explains what information needs to be extracted from a knowledge base and what problems are encountered. In a first step we need to find all classes and entity types. After that we need to find all connections between classes and the corresponding properties. Furthermore, it is desirable to find out how much the classes and properties of the knowledge base are used. Using this information we display the most used classes and properties to the user and make it easier to gain an insight into the focus of the knowledge base.

Query Group 1: Type Queries

```
SELECT DISTINCT ?c WHERE { ?s a ?c . }
SELECT DISTINCT ?c WHERE {?s <p> ?o . ?s a ?c . }
SELECT DISTINCT ?c WHERE {?s <p> ?o . ?o a ?c . }
```

Query Group 2: Property Queries

```
SELECT DISTINCT ?p WHERE { ?s ?p ?o . }
SELECT DISTINCT ?p WHERE { <c> ?p ?o . }
```

A logical first step is to request all used classes or properties in a knowledge base (using the first queries in Query Group 1 and 2). Note that in order to classifiy also SPARQL 1.0 endpoints, we did not use the EXISTS filters. A SPARQL endpoint will possibly not answer to these queries, depending on the size and complexity of the knowledge base. For example, if we request BNF [1] using the first query from Query Group 2 it results in a server error. The reason for this is the performance of the underlying server and the size and complexity of the knowledge base. To further analyze such problems we define four types of knowledge bases: light, type-heavy, property-heavy and heavy knowledge bases

A light knowledge base is a light data set that can answer all queries from query group 1 and 2. Note that <c> and <p> represent classes or properties, contained in the knowledge base. It is important, that the endpoint can answer for all values for <c> and <p> in order to yield as a light knowledge base. We did not use the $EXIST$ filter because we wanted to include SPARQL 1.0 endpoints in our definition/classification.

The reason that the endpoint is able to answer the queries, may be due to the power of the server, an index optimized for such queries, or a data set with few properties and classes.

A type-heavy knowledge base is a data set that cannot respond to all queries presented in Query Group 1. This may be because the server has too few resources, the index is not optimized for such a query, too many types are used in the data set or simply because the data set is very huge.

Similar to type-heavy, a property-heavy knowledge base cannot respond to the queries shown in Query Group 2.

Knowledge bases that are both, type-heavy and property-heavy, are denoted as heavy knowledge bases.

Schema of the first three types can still be derived rather easily, because in the case of type-heavy we can simply query all properties and then query the source and object classes for each property. This reduces the amount of results and the server is not stressed as much. In the case of property-heavy, the procedure is exactly the other way around. First all classes or types are queried and then the properties for each class in the knowledge base are queried. In a last step we have to test which classes are connected to each other.

In principle, we can use approaches as presented by the LODeX System [12, 11, 6, 15] to derive a schema. However, with heavy knowledge bases we encounter the problem that we can not use any of the procedures described above. Both procedures result in a server error even when using the LODeX algorithms.

Therefore, our goal is to find a way, to infer the schema of heavy knowledge bases like DBpedia [4] or WikiData [10].

4.2 Connecting Linked Data and Web APIs

If a user wants to use multiple heterogeneous data repositories with heterogeneous data modalities (e.g. SPARQL endpoints and Web APIs), it is important to have some information of these data endpoints. For example, if we want to integrate the SciGraph Web API [9], it is necessary to know the URL to address the API and which parameter the API requires. In case of the previous mentioned Web API we can use three different parameters to create a valid HTTP request[9]: these parameters are used to request information about a publication by using (1) the DOI of a paper, (2) the ISBN of a book (3) or the ISSN of a journal. The goal of this part is to learn the appropriate input types to the corresponding Web API, e.g. DOIs, ISBNs or ISSNs.

Therefore, we need to learn the *configuration of a data endpoint*, i.e. to learn what kind of values the parameters of the Web API expect. Consider the example of Springers SciGraph Web API, it requires a parameter called "doi". It is a big overhead, if the user of a federated system has to test every value of a knowledge base in order to determine the correct configuration for a Web API. For this reason, an automatic interface detection for SPARQL is designed and implemented. This detection algorithm uses different techniques to match the parameters with the corresponding data types it can process, e.g. DOIs.

Similar as in ANGIE, Web APIs are modeled as RDF graph and describe which input parameters are required or are optional. The difference to ANGIE is that in our approach a user do not have to specify which data types belong to what parameters (e.g. *?id* takes DOIs as input values). Only the parameters need to be specified and afterwards the system determines the appropriate data types itself. Furthermore, the graph stores the linkage points between the Web API and the knowledge base. Using this information, we can later determine what API needs to be requested to fill the knowledge base with missing information. To realize this detection algorithm we perform the following steps:

In a first step we match the parameter names with the property names of our knowledge bases, e.g. *doi* and *dblp:doi*. We do this so that we do not have to test all properties from our knowledge base with the Web API parameters, e.g. *dblp:title* and *doi* or *dblp:isbn* and *doi*. This is only a small improvement, since most Web API parameters do not have a clear meaning like *q* or *id*. We can not simply match parameters like *q* with the properties in our database, because this labeling is too general.

If we can match the API parameter names with properties from our knowledge base, we send in the next step some requests to the Web API in order to check if we get results. Therefore, we select from the found properties a number of α randomly selected entities and send these values with the corresponding parameters to the Web API. For example, in case of SciGraph [9] we can get $\alpha = 25$ DOIs from our knowledge base and send them to the Web API using the corresponding parameter *doi*.

If we can not match the API parameter names with properties from our knowledge base, we need to do the above described procedure for all properties in our knowledge bases. If we consider WikiData[10] as knowledge base, we have several hundreds of properties that need to be checked. Therefore, the first step reduces the search space considerably and excludes some properties from the beginning.

In the next step we need to check, whether the Web API responses with meaningful data. Some Web APIs have a fuzzy search and, in doubt, return any or the best matching result before they return none. For this reason we define a meaningful response in the following way: *A meaningful response consists of an amount of data in which we already know a minimum amount of information.* This means, that the information returned needs to overlap with the information in our knowledge base, in order to measure that the Web API returned a valid response and not just the best matching result.

But before we can determine whether the responses we receive are meaningful, we need to send requests to the Web API again. This time we only send property data to the

Web API for which we got a response. In the previous step, we only sent a small number (denoted by α) of requests to the Web API to see if we get an answer. This time we need more data/responses in order to evaluate correctly if we get meaningful answers. For this reason we send a number of β requests per property to the Web API.

To test whether some information in the response matches our data, we need to use record linkage algorithms and metrics. Since most Web APIs send JSON or XML as response, we first need to transform this response in a Linked Data format. Because both, JSON and XML, represent tree structures, we can in a first step flatten the tree. The URL, in which the query for the Web API is encoded, is used in RDF format as subject. The path of the flattened JSON/XML response to the actual values are used as properties and the values are used as objects in the created RDF format.

Afterwards we can evaluate, rather the created RDF response is meaningful. Only when a minimum amount (denoted by γ) of information overlaps we can be sure that we have received a meaningful response.

As it should be clear, the choice of the thresholds α, β and γ is critical when it comes to the quality of the matching and the run time. In addition, there is a new combination of these three values for each selected pair of knowledge base and Web API. In order to get the best results we need to identify the optimal combination of these three values.

It is also possible to change the focus of the matching by varying the threshold values. For example, you can achieve an exact matching by using high threshold values and also reduce the number of requests by using low threshold values and few requests and thus it is possible to find matches even for paid Web services.

To present linkage points between the knowledge base and the Web API to the user, we provide a visual representation of the derived schema and its linkage points to the corresponding Web API.

In order to prevent the visualization from becoming cluttered, we combine for large knowledge bases, e.g. WikiData, all classes in their super classes and provide in addition the possibility to only show the most used classes and connections in the knowledge base. This allows a non-expert user to quickly see which information can be added to the knowledge base. Accordingly, a schema of the Web API must also be created on record level.

4.3 Future Prospects

In this section, we describe briefly planned work that we have not yet been able to devote ourselves to.

User Defined Functions

As already explained, we want to extend SPARQL and give an expert user the possibility to implement domain specific functions such as a similarity search and use it in SPARQL. We use Web APIs again, because they are very flexible and easy to program. In addition, developers are not limited to the choice of a single programming language and can make Web APIs easily accessible to a community.

By using the SERVICE and BIND keywords from SPARQL we want to call this user defined functions and bind the output to variables. This approach is also used by the Ephedra system [24, 23] and the Authors stated that the adaption effort is a complex task and needs to be minimized. To do so, we want to store the Web API as a function graph in a triple store. The graph should provide information about what kind of parameters the function has, whether they are optional, and what kind of output is provided.

Our goal is that a user can call this custom functions using the data from our Hybrid knowledge base. This includes the data from Web APIs, e.g. SciGraph [9] or CrossRef [2].

Visual Query Interface

In the future, we want to develop a visual query interface (as already proposed in LODeX [13, 14], FacetGraphs [18] and many others). Our goal is to create an interface, which is easy to use for non-expert users but also has powerful functions from SPARQL such as filters, groupings, orderings and so on. Furthermore, we want to integrate the previous described user defined functions into the visual query interface, so that non-experts can use domain specific functions shared by developers. This part is not particular novel, but serves as an intermediate step to determine the difficulty of integrating Web APIS with SPARQL Endpoints. In addition, we will use this interface to evaluate how well non-expert users can work on hybrid federations and which issues arise and need to be fixed.

5. EVALUATION PLAN

Our goal is that we can extract the schema of a heavy knowledge base (see Section 4.1) as precisely as possible. Furthermore, we want to link information from Linked Data knowledge bases to the data from Web APIs. In order to show the soundness of our approach, we will describe in the following our evaluation plan.

5.1 Evaluating Schemas

To determine whether the extraction of the schema worked correctly, we need to create two types of data sets. The first type of data set is intended to test and train the schema extraction algorithm. The second type of data set is used to evaluate the correctness of the algorithm. To prevent us from falsifying the evaluation, we decided to use two types of data sets.

The first step to evaluate the schema inference algorithm is to extract the schema of an endpoint by hand (this will be the used gold standard). This implies that the data is searched manually and the schema of the endpoint is extracted accordingly. The Schema Inference algorithm is then applied to the endpoint. The final step is to compare the two derived schemas and measure how similar they are. Therefore, we will store both schemas in a triple store, using RDF, and count how many triples are common or missing. In case of heavy endpoints, it is hardly possible to derive the schema manually, which is why this type of evaluation is not suitable here. Instead, we will test the correctness of our procedures on light and property-heavy endpoints and, for heavy endpoints, only test them in random samples. This means that we will check whether the connections derived in the schema exist or not. This will not help to find missing connection but additional connections that erroneously are derived and actual do not exist in the knowledge base.

5.2 Evaluating Automatic Datatype Detection

When it comes to connecting Web APIs to Linked Data endpoints and forming a hybrid federation, two different parts need to be evaluated.

First, we need to check which data types were recognized for the Web APIs, whether they were the correct ones, and whether all matching data types were found. On the other hand we have to check if the record linkage worked correctly. Here, too, we want to divide Web APIs into two groups, as we did previously with the evaluation of the schema. The first group of Web APIs is used again to test and adapt the algorithms used. The second group of Web APIs is again used to evaluate and verify the algorithms used.

5.2.1 Data Type Detection

The first step in evaluating the data type detection algorithm is to find all data types from the knowledge base that should be classified as correct parameters for a specified Web API. As in the case of the evaluation of the Schema Inference algorithm, this step must be performed manually for the first time and serves as the gold standard. The data types of the gold standard can then be compared with the found data types of the Data Type Detection Algorithms to evaluate the used algorithm.

5.2.2 Record Linkage

As already mentioned, it must also be evaluated how good the results of the record linkage is. An overview of data linkage is presented in [16]. The authors recommend to use precision-recall or F-measure graphs rather than single numerical values to measure the quality of linkage algorithms. Data pairs that should not be matched because they are not identical are called true negatives. Quality measure that include the number of true negative matches should not be used due their large number in the space of record pair comparisons, otherwise they would falsify the evaluation.

6. ACKNOWLEDGEMENT

A special thanks goes to my supervisor Ralf Schenkel for his invaluable support.

7. REFERENCES

[1] BNF Bibliothèque nationale de France. `http://www.bnf.fr/`.

[2] CrossRef. `https://www.crossref.org/services/metadata-delivery/rest-api/`.

[3] dblp computer science bibliography. `https://dblp.uni-trier.de`.

[4] DBpedia. `https://wiki.dbpedia.org`.

[5] FOAF vocabulary specification 0.99. `http://xmlns.com/foaf/spec/`.

[6] LODeX Model. `http://dbgroup.unimo.it/lodex_model/lodex`. Accessed: 27.02.2019.

[7] RDF Schema 1.1. `https://www.w3.org/TR/rdf-schema/`. Accessed: 27.02.2019.

[8] SPARQL query language for rdf. `https://www.w3.org/TR/rdf-sparql-query/`. Accessed: 27.02.2019.

[9] Springer SciGraph Web API. `https://scigraph.springernature.com/explorer/api/`.

[10] WikiData. `https://www.wikidata.`

[11] F. Benedetti, S. Bergamaschi, and L. Po. A visual summary for linked open data sources. *CEUR Workshop Proceedings*, 1272:173–176, 2014.

[12] F. Benedetti, S. Bergamaschi, and L. Po. Online index extraction from linked open data sources. *CEUR Workshop Proceedings*, 1267(January):9–20, 2014.

[13] F. Benedetti, S. Bergamaschi, and L. Po. LODeX: A tool for visual querying linked open data. *CEUR Workshop Proceedings*, 1486:2–5, 2015.

[14] F. Benedetti, S. Bergamaschi, and L. Po. Visual Querying LOD sources with LODeX. pages 1–8, 2015.

[15] F. Benedetti, S. Bergamaschi, and L. Po. Exposing the underlying schema of LOD sources. *Proceedings - 2015 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology, WI-IAT 2015*, 1:301–304, 2016.

[16] P. Christen and K. Goiser. Quality and complexity measures for data linkage and deduplication. pages 127–151, 2007.

[17] O. Görlitz and S. Staab. SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. 2011.

[18] P. Heim, T. Ertl, and J. Ziegler. Facet graphs: Complex semantic querying made easy. pages 288–302, 2010.

[19] K. Kellou-Menouer and Z. Kedad. On-line Versioned Schema Inference for Large Semantic Web Data Sources. *Proceedings of the 29th International Conference on Scientific and Statistical Database Management - SSDBM '17*, 2017.

[20] M. Konrath, T. Gottron, S. Staab, and A. Scherp. Schemex - efficient construction of a data catalogue by stream-based indexing of linked data. *J. Web Semant.*, 16:52–58, 2012.

[21] M. Koutraki, D. Vodislav, and N. Preda. Deriving intensional descriptions for web services. pages 971–980, 2015.

[22] F. H. M. Weise, S. Lohmann. LD-VOWL: extracting and visualizing schema information for linked data endpoints. 2016.

[23] A. Nikolov, P. Haase, J. Trame, and A. Kozlov. Ephedra: Efficiently combining RDF data and services using SPARQL federation. 786:246–262, 2017.

[24] A. Nikolov, P. Haase, J. Trame, and A. Kozlov. Ephedra: SPARQL federation over RDF data and services. 2017.

[25] N. Preda, F. M. Suchanek, G. Kasneci, T. Neumann, M. Ramanath, and G. Weikum. ANGIE: active knowledge for interactive exploration. *PVLDB*, 2(2):1570–1573, 2009.

[26] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: A federation layer for distributed query processing on linked open data. pages 481–486, 2011.

[27] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: Optimization techniques for federated query processing on linked data. pages 601–616, 2011.

[28] B. Stringer, A. Meroño-Peñuela, S. Abeln, F. van Harmelen, and J. Heringa. SCRY: extending SPARQL with custom data processing methods for the life sciences. 2016.