KRaider: a Crawler for Linked Data

Giuseppe Cota[1 [0000−0002−3780−6265]], Fabrizio Riguzzi[2 [0000−0003−1654−9703]], Riccardo Zese[1 [0000−0001−8352−6304]], and Evelina Lamma[1 [0000−0003−2747−4292]]

[1] Dipartimento di Ingegneria – Università di Ferrara
[2] Dipartimento di Matematica e Informatica – Università di Ferrara
Via Saragat 1, 44122, Ferrara, Italy
{giuseppe.cota,fabrizio.riguzzi, riccardo.zese,evelina.lamma}@unife.it

Abstract. The aim of the Semantic Web and Linked Data principles is to create a web of data that can be processed by machines. The web of data is seen as a single globally distributed dataset. During the years, an increasing amount of data was published on the Web. In particular, large knowledge bases such as Wikidata, DBPedia, LinkedGeoData, and others are freely available as Linked Data and SPARQL endpoints. Exploring and performing reasoning tasks on such huge knowledge graphs is practically impossible. Moreover, triples involving an entity can be distributed among different datasets hosted by different SPARQL endpoints. Given an entity of interest and a task, we are interested into extracting a fragment of knowledge relevant to that entity, such that the results of the given task performed on the fragment are the same as if the task was performed on the whole web of data.
Here we propose a system, called KRaider ("Knowledge Raider"), for extracting the relevant fragment from different SPARQL endpoints, without the user knowing their location. The extracted triples are then converted into an OWL ontology, in order to allow inference tasks. The system is part of a - still under development - framework called SRL-Frame ("Statistical Relational Learning Framework").

Keywords: Linked Data · SPARQL · Ontology · RDF · Semantic Web.

1 Introduction

The aim of the Semantic Web and Linked Data principles [2] is to create a web of data that can be processed by machines. During the years, an increasing amount of data was published on the Web. In particular, large knowledge bases such as Wikidata [21], DBPedia [12], LinkedGeoData [18], and others are freely available. These knowledge bases are represented with Semantic Web standards like RDF and OWL. They contain thousands of classes and properties, and millions of triples. Moreover, new small knowledge bases and HTML pages with semantic content are continuously published.

Published data is generally available as data dumps, Linked Data documents, Triple Pattern Fragments or SPARQL Endpoints. Moreover, according to the Linked Data principles, triples involving an entity can be distributed among

different datasets. All this data constitutes the web of data, that can be seen as a single globally distributed dataset.

Due to its sheer size, it is difficult to explore or perform complex tasks such inference tasks (e.g. inconsistency checks) on the whole Web of data. Reasoners like Pellet [17], Hermit [16] and BUNDLE [4,15] can handle relatively small knowledge bases. A better scalability of these algorithms can be achieved by taking into account only the "interesting parts" of the web of data. In particular, given an entity of interest, we are interested into extracting the fragment of knowledge relevant to that entity from the Web of data.

In this paper we propose a system, called KRaider ("Knowledge Raider"), for extracting the relevant fragment from different SPARQL endpoints, without the user knowing their location. The extracted triples are then converted into an OWL ontology, in order to allow inference tasks. The system is part of a - still under development - framework called SRL-Frame ("Statistical Relational Learning Framework").

The paper is organized as follows. Section 2 provides an overview of the main interfaces used to obtain knowledge from the Web of data. Section 3 illustrates the problem of the extraction of the knowledge fragment relevant to an entity. Section 4 presents the system KRaider and the framework SRL-Frame. An evaluation of KRaider used in SRL-Frame is provided in Section 5. Finally, Section 6 draws conclusions.

2 Related Work

In this section we provide a brief overview of the main paradigms and interfaces for publishing and querying Linked Data based on RDF triples:(i) data dumps, (ii) Linked Data Documents, (iii) SPARQL endpoints, and (iv) Triple Pattern Fragments.

In particular, for each interface, we illustrate how queries can be performed by clients and/or servers using the SPARQL language, which is the standard language defined by the W3C for expressing queries on RDF triples.

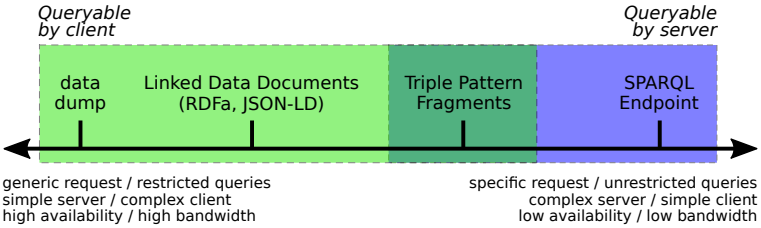


Fig. 1: Interfaces for querying Linked Data.

Figure 1 shows an overview of the main interfaces for publishing and querying Linked Data. A SPARQL endpoint allows clients to execute SPARQL queries

on RDF triples, whereas a server that publishes data dumps or Linked Data Documents does not provide a way for executing queries, which is, instead, delegated to clients. Moreover, there exist hybrid approaches like Triple Pattern Fragments [20] that divide the query workload between servers and clients.

2.1 Data Dumps

The easiest way to publish Linked Data is to upload to a server an archive (data dump) containing one or more files in an RDF syntax such as Turtle or N-Triples. The client downloads the data dump, extracts the contained files and processes them for performing queries.

The main advantage is that the server is easy to maintain and the requests received from clients can be easily handled, i.e. the complexity of the server is low.

However, this approach has several disadvantages. First of all, even if only few triples are needed, the client has to download the whole dump and a high bandwidth may be necessary. Second, if part of the data becomes outdated, the client has to download again the archive. Moreover, the cost for managing data can be very high for clients. For huge RDF graphs, the client is required to perform data-intensive tasks in order to answer queries.

Data centralization approaches aim at providing support for query execution over a collection of RDF data harvested from several sources. LOD Laundromat [1] is one of them: it gathers data dumps in various RDF formats, cleans up the data by removing syntax errors, duplicates and blank nodes, and then converts and re-publishes the collected dumps into RDF-compliant formats. Moreover, LOD Laundromat allows querying the datasets using approaches based on Triple Pattern Fragments (see subsection 2.4).

If a query service is provided, a data centralization system can provide fast responses. However, if the central repository server is not available, all the clients that rely on it are unable to function. Therefore, in order to avoid unavailability (and latency), clients still have to download the needed data dumps. Furthermore, even without any query service, the cost of maintaining a centralized repository can be very high. Finally, given the dynamic nature of Linked Data, the collected dumps may be outdated.

2.2 Linked Data Documents

RDF triples are divided into several Linked Data documents organized by entity. Typically, each document d_e consists of triples related to the entity e identified by an URI, where the subject of the triples is that entity, i.e. d_e contains triples of the form (e, p, o).

These approaches follow the Linked Data principles by Tim Berners-Lee [2]. In fact, if a client does not know what an entity identified by an URI represents, it can find information about that entity by "dereferencing" its URI (usually by means of an HTTP GET request). For instance, the entity Leonardo da Vinci is

denoted by the URI `http://dbpedia.org/resource/Leonardo_da_Vinci`, and dereferencing this URI leads to a document containing triples in which the URI is the subject.

Linked Data documents can be represented in any RDF syntax. In particular, due to the popularization of REST architectures for web applications, an RDF format based on JSON, called JSON-LD was developed. Moreover, in order to reduce the effort for producing documents, RDFa was proposed, which allows publishing Linked Data in HTML5 documents.

From the viewpoint of the server, the required cost to generate each document is low and the performances for responses can be high. Moreover, the same document can be reused by many clients, allowing the sever to apply cache reuse policies to reduce the response time.

In [8], Hartig surveyed the approaches to execute queries over Linked Data documents. These approaches can be split into two main categories. One category uses pre-populated index structures [19]. The other one performs link traversal to dynamically discover data for answering the query [9]. Link traversal approaches have usually long query execution times. However, they require less bandwidth than data dumps and, unlike data dumps, the data used to answer the query is up-to-date. The major drawback of these methods is that the completeness of the answers with respect to a knowledge graph cannot be guaranteed [7]. In particular, queries that contain triple patterns with unbound subject may cause some issues. For instance, the following query is not Linked Data-answerable:

```
SELECT ?entity WHERE { ?entity foaf:name "Leonardo da Vinci" }
```

2.3 SPARQL Endpoints

The most straightforward way for a client to execute a SPARQL query is to send the query to the server and delegate the entire execution to the server, which sends the answer back to the client. A SPARQL endpoint enables the clients to execute queries on a dataset through HTTP.

Although allowing clients to submit arbitrary SPARQL queries leads to low bandwidth consumption and low client cost, the cost of processing the whole query server-side may be really high, for the server, in terms of CPU time and memory consumption. In fact, evaluating a SPARQL query is PSPACE-complete [14]. Another disadvantage of SPARQL endpoints is that client queries are highly individualized. Therefore, caching the results does not lead to significant improvements.

In order to reduce the computational cost of evaluating SPARQL queries, many endpoints use fragments of SPARQL with less expressive power (and hence lower complexity), reduce the allowed query execution time and limit the number of rows that can be returned (for instance, the SPARQL endpoint of DBPedia [12] has a limit of 10,000 rows).

The idea of *query federation* is to answer queries based on information from many different sources. SPARQL 1.1 from the SPARQL W3C working group added the SERVICE operator to the language specification, which can be used for querying another remote SPARQL endpoint during query execution.

2.4 Triple Pattern Fragments

The Triple Pattern Fragment [20] (TPF) interface aims at reducing query execution costs by moving part of the execution workload from servers to clients.

A fragment consists of all triples that match a specific triple pattern (plus metadata and controls). SPARQL queries are decomposed by clients into triple pattern queries, i.e. queries composed of a single triple pattern. The server is only responsible for providing solutions to triple patterns, i.e. providing the fragment of a given triple pattern query. The client is responsible of combining the obtained fragments by processing operators such as join, union and optional.

An example of a triple pattern is: `dbpedia:Donald_Duck ?p ?o`. The formal definition of triple pattern is reported below.

Definition 1. *Let $\mathcal{V}$ be the infinite set of variables, which is disjoint from the set $\mathcal{U}$ of all URIs and the set $\mathcal{L}$ of all literals. Any triple $tp \in (\mathcal{V} \cup \mathcal{U} \cup \mathcal{L}) \times (\mathcal{V} \cup \mathcal{U}) \times (\mathcal{V} \cup \mathcal{U} \cup \mathcal{L})$ is a triple pattern.*

Since queries received by a server are less specific, it is more probable that the same fragment may be reused by multiple clients. Therefore Triple Pattern Fragments can exploit caching. This interface allows servers to maintain high availability and to scale to a much larger number of clients. However, it is not flawless. First of all, this interface requires the maintenance of dedicated servers and clients. Moreover, the required bandwidth is much higher. In fact, a query is decomposed into multiple triple pattern queries leading to a large amount of data transferred between servers and clients. In order to overcome this issue, Hartig and Buil Aranda [10] proposed an extension of TPF that allows clients to send to the server bindings in the queries in addition to triple patterns. There exist hybrid approaches that combine SPARQL endpoints and Triple Pattern Fragments, in order to exploit the advantages of both paradigms [13].

3 Extraction of Knowledge Fragments

In [11], the authors defined an approach to extract a knowledge fragment relevant to an entity in order to perform a learning task. This approach is integrated into DL-Learner [3] and the fragment can be extracted by using only one SPARQL endpoint. Moreover, the user must know the URL of the endpoint.

In this paper, in order to enable the users to perform complex tasks such as inference on Linked Data, we propose KRaider, a system for discovering relevant knowledge fragments. The approach is similar to the one proposed in [11], but, unlike [11], it can extract knowledge from multiple SPARQL endpoints by using an approach inspired by Link Traversal methods [9].

Below we provide the definition of desired knowledge fragment relevant to an entity.

Definition 2. *The desired knowledge fragment F_e relevant to an entity e is the smallest fragment of the Web of data $\mathcal{W}$ ($F_e \subset \mathcal{W}$), such that a task involving entity e and performed on F_e, provides the same results as if the task was performed on $\mathcal{W}$.*

In other words, we want to extract a fragment that holds enough information and that is small enough to allow the efficient execution of various tasks.

The relevant fragment is extracted by recursively traversing the RDF graphs starting from an entity. The recursion depth is a parameter that can be set by the user and affects the size of the extracted fragment.

The extraction algorithm starts from an entity e identified by an URI U_e, then extracts the triples that have e as subject, i.e. triples of the form (e, p, o). The recursion depth is decremented and the objects o of the obtained triples are used to extract additional knowledge until the user-defined recursion depth is reached. In case p is equal to `owl:sameAs`, the recursion depth is not decremented. Moreover, if the object's URI U_o has a domain D_o which is different from the domain of subject's URI, in the next recursive step, the SPARQL endpoint hosted by D_o is also queried.

The object of the obtained triples are used to extract additional knowledge until the user-defined recursion depth is reached. The fragment extraction process is shown in Figure 2.

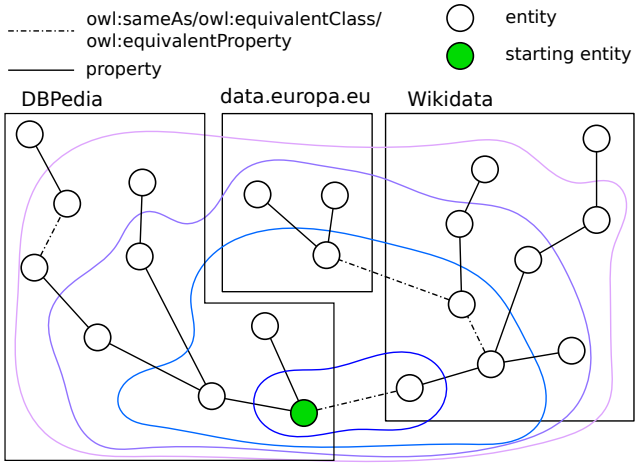


Fig. 2: Extraction of the relevant fragment from different SPARQL endpoints (Wikidata, data.europa.eu and DBPedia). The closed curves represents different recursion depths. The smallest curve represent the fragment with recursion depth 0, the larger inner curve represents the fragment with recursion depth 1, up to the largest outer curve with recursion depth 3. It should be noted, that the predicate `owl:sameAs` does not increment the recursion depth.

4 Extraction Framework

In this section we provide the conceptual and some technical details about the framework used to extract the relevant knowledge fragment.

4.1 Architecture

The system that takes care of extraction of the knowledge fragment from SPARQL endpoints is KRaider, which is integrated into SRL-Frame, a framework under development written in Java and based on the OSGi technology. In particular, it uses Apache Felix as the OSGi implementation.

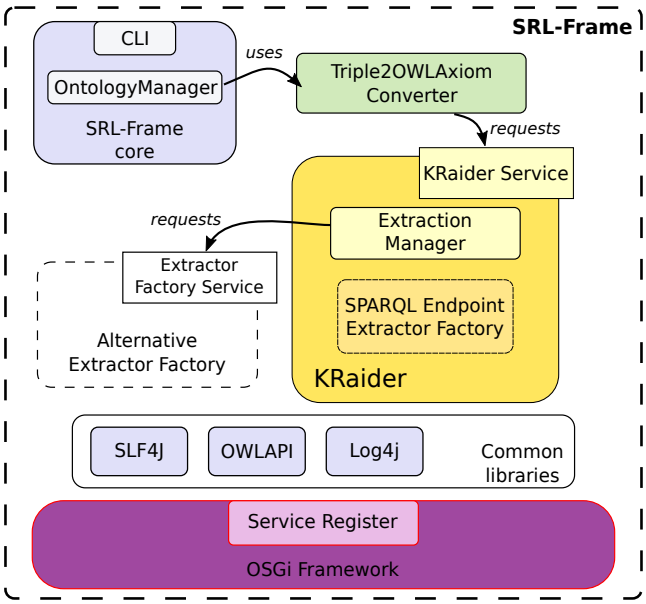


Fig. 3: Architecture of SRL-Frame.

The architecture of SRL-Frame is shown in Figure 3. The framework provides some common libraries like SLF4J and Log4j for logging and OWL API for manipulating ontologies. KRaider is a bundle[3] and a service of this framework. When SRL-Frame is launched, an instance of (an implementation of) KRaider is registered into the OSGi Service Register. Therefore, if another bundle wants to use a KRaider instance, it should check if there is one in the Service Register. The OSGi technology is useful to enable or disable services on the fly. In fact, if for some reason the KRaider service was disabled (e.g. maintenance, replacement of the service with a better implementation), the client bundle does not find KRaider in the Service Register and it simply does nothing.

The implementation of KRaider provides a factory for building new *extractors* (SPARQLEndpointExtractorFactory). An extractor is a component that *extracts* new triples from the Web of data. In particular, a SPARQL endpoint extractor is a component that can find new triples by exploiting SPARQL

[3] In the OSGi jargon a bundle is a module.

endpoints. When an extractor is created, it is assigned to a new thread by a component called ExtractionManager, which manages the created extractors. In particular, it is responsible for the creation of new extractors, handles the extracted triples and assigns jobs to extractors (see subsection 4.2 for further details).

The ultimate goal of KRaider is to be able to use several types of extractors, each exploiting a different Linked Data interface. The service oriented philosophy of OSGi comes in handy to realize this. In fact, KRaider can check, at run-time, which extraction services are available and then choose the ones to use. However, at the moment, KRaider only contains a single type of extractor which is able to exploit SPARQL endpoints to obtain triples. No other kinds of extractors have been implemented yet.

4.2 Triple Extraction with KRaider

Algorithm 1 shows the algorithm of ExtractionManager. Its interactions with the extractors are graphically summarized in Figure 4.

Given a recursion depth, an URI that represents an entity e, a list of available SPARQL endpoints and a triple queue as input, ExtractionManager extracts the domain from the given URI[4] (line 7), creates the first job, where a job is a quadruple of the form $\langle U_e, O_e, S_e, D_e \rangle$, where U_e is the URI of entity e, O_e is its domain, S_e is the status, and D_e the recursion depth (line 8). The status can have four possible values: *Available*, *Running*, *NotAnswerable* and *Complete*. Then it checks if new extractor threads can be launched and listens for extractor requests.

When an extractor requests a new job (line 11), it also send the set of extracted triples and the job j that has just been completed (in the first request the list will be empty and the completed job will be null). The extracted triples will be all those triples that have as subject the entity represented by the URI in job j.

If the extractor was able to extract new triples, it sends a requests where the computed job's status is *Complete*, otherwise it sends an empty set of triples and the computed job's status is *NotAnswerable*. Then the manager takes the first available job, i.e. a job with status equal to *Available*, sends it to the extractor (lines 12-14) and sets the job status to *Running*. If there are no available jobs at the moment, the manager stops the extractor that performed the request.

When the manager receives a request from the extractor together with a non-empty set of newly extracted triples and the computed job is $\langle U_s, O_s, S_s, D_s \rangle$, for each new triple of the form (s, p, o), if o is a named resource, it extracts the domain O_o from U_o and adds the following job to the job queue:

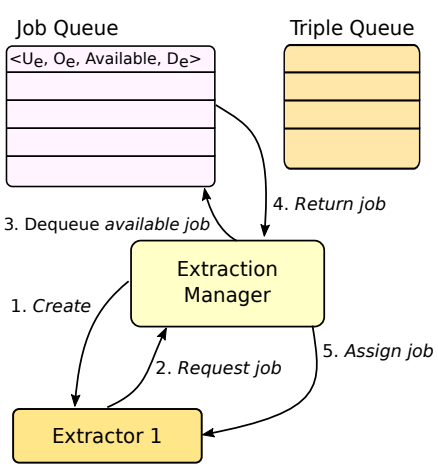
$$\langle U_o, O_o, \mathit{Available}, D_o \rangle$$

In addition, if the domain O_s is different from O_o, then it means that another SPARQL endpoint should be taken into account and the manager adds the

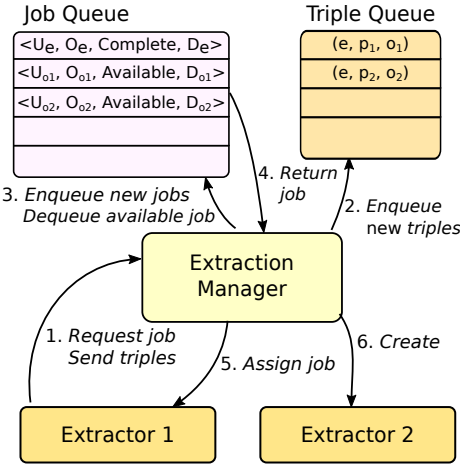
[4] We assume that each domain corresponds to a SPARQL endpoint.

Algorithm 1 KRaider's ExtractionManager

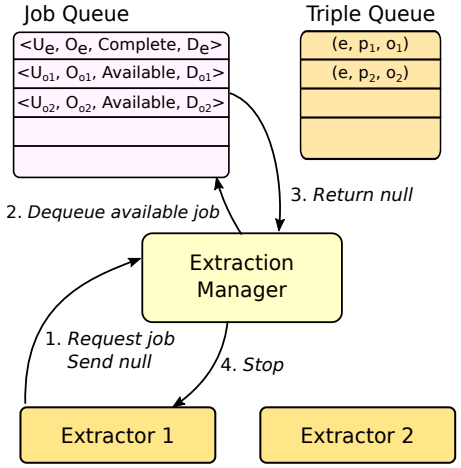
```
1: function ExtractionManager(U_e, D, E, T)
2:     Input: URI representing an entity U_e
3:     Input: max recursion depth D
4:     Input: list of available SPARQL endpoints E
5:     Input: triple queue T
6:     Create job queue J
7:     O_e ← ExtractDomain(U_e)
8:     Enqueue job ⟨U_e, O_e, Available, D_e⟩ into J
9:     do
10:        StartExtractors(J, E) ▷ Start extractor threads according to available jobs
11:        T, j ← WaitRequests() ▷ Listen to requests and receive from each extractor the set of triples T and the computed job j
12:        l ← DequeueAvailable(J)
13:        Update l status to Running
14:        Send l to extractor
15:        Enqueue triples T into T
16:        for all (s, p, o) ∈ T do
17:            if o is a named resource then
18:                O_o ← ExtractDomain(U_o)
19:                if p ≠ owl:sameAs then
20:                    D_o = D_s − 1
21:                else
22:                    D_o = D_s
23:                end if
24:                Enqueue job ⟨U_o, O_o, Available, D_o⟩ into J
25:                if O_s ≠ O_o then
26:                    Enqueue job ⟨U_o, O_s, Available, D_o⟩ into J
27:                end if
28:            end if
29:        end for
30:    while AvailableJobs(J)
31: end function
```



(a) First operations of KRaider.



(b) Dynamic creation of new extractors when new jobs are available.



(c) Termination of an extractor caused by the lack of available jobs.

Fig. 4: KRaider's fragment extraction process.

following job (where $O_o \neq O_s$):

$$\langle U_o, O_s, Available, D_s \rangle$$

Moreover, if p is different from `owl:sameAs`, the recursion depth is decreased $D_o = D_s - 1$, otherwise it is left unchanged (lines 16-29).

Finally, if there are no available jobs the algorithm terminates.

The algorithm SPARQLEndpointExtractor which extracts triples from a SPARQL endpoint is shown in Algorithm 2. The extractor gets as input a list of the SPARQL endpoints $\mathcal{E}$ and a job $\langle U_s, O_s, S_s, D_s \rangle$. It checks if there is a SPARQL endpoint hosted by domain O_s (line 5) In the list of the available endpoints $\mathcal{E}$. If no such endpoint exists, it updates the job status to *NotAnswerable* (line 24) and request a new job to ExtractionManager (line 26). Otherwise, it queries the endpoint 3 times in order to obtain the list of URIs U_{e_i} (with $i = 1 \dots n$) of the entities which are the same as (`owl:sameAs`) or equivalent (`owl:equivalentClass`,
`owl:equivalentProperty`) to U_s (lines 8-19). Then it sends the following query to the endpoint (line 20:

```
SELECT DISTINCT * WHERE {
 { <U_s> ?p ?o . } UNION
 { <U_e1> ?p ?o . } UNION
 ... { <U_en> ?p ?o . } }
```

The triples that bind with the union of the triple patterns of the query are extracted.

4.3 OWL Conversion of the Extracted Fragment

Each extracted triple is added by ExtractorManager to a triple queue (TripleQueue). Many reasoners like Pellet [17], Hermit [16] and BUNDLE [4,15] are able to perform inference on OWL ontologies. For this reason, we developed a conversion pipeline which converts the extracted triples into an OWL ontology. Figure 5 shows how the conversion is performed.

The triples in TripleQueue are repeatedly dequeued by a converter (Triple2OWLAxiomConverter), which converts each triple into an OWL axiom and then enqueues the axiom into another queue called OWLAxiomQueue. This queue is consumed by OntologyManager, which annotates the OWL axioms with their origins, i.e. with the dataset from which an axiom was extracted, and enqueues them to an OWL ontology. Finally, if a timeout was reached or the triple queue doesn't contain any triples (that means that KRaider stopped), the converter and the ontology manager stop.

The conversion from RDF to OWL is performed by following the mapping defined by the W3C in [6]. Moreover, for triples of the form (s, `rdf:type`, o), if s or o are classes, the predicate `rdf:type` is converted to `rdfs:subClassOf`.

Algorithm 2 SPARQLEndpointExtractor

function SPARQLEndpointExtractor($\mathcal{E}$, $\langle U_s, O_s, S_s, D_s \rangle$)
    Input: a job $\langle U_s, O_s, S_s, D_s \rangle$
    Input: a list of SPARQL endpoints $\mathcal{E}$
    $j \leftarrow \langle U_s, O_s, S_s, D_s \rangle$
    $E \leftarrow$ GetEndpoint(O_s, $\mathcal{E}$) ▷ get from $\mathcal{E}$ the SPARQL endpoint that is hosted by domain O_s
    $T \leftarrow$ *emptySet*
    if $E \neq$ *null* **then**
        $S \leftarrow$ GetSameAsIndividuals(U_s)
        for all $U_e \in S$ **do**
            Add (U_s, `owl:sameAs`, U_e) to T
        end for
        $E \leftarrow$ GetEquivalentClasses(U_s)
        for all $U_e \in E$ **do**
            Add (U_s, `owl:equivalentClass`, U_e) to T
        end for
        $P \leftarrow$ GetEquivalentProperties(U_s)
        for all $U_e \in S$ **do**
            Add (U_s, `owl:equivalentProperty`, U_e) to T
        end for
        $Q \leftarrow$ BuildQuery(j, S, E, P)
        $T \leftarrow$ SPARQLQuery(Q)
        $j \leftarrow \langle U_s, O_s, Complete, D_s \rangle$
    else
        $j \leftarrow \langle U_s, O_s, NonAnswerable, D_s \rangle$
    end if
    RequestJob(T, j)
end function

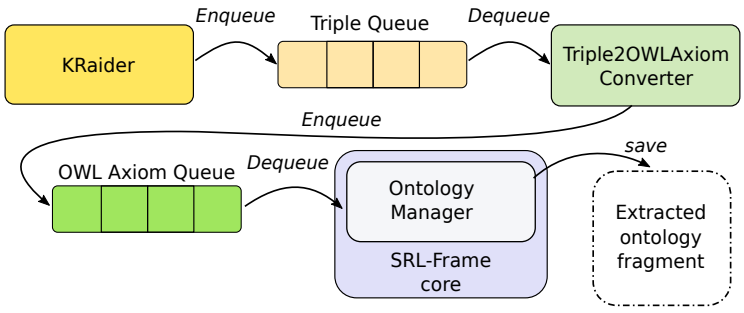


Fig. 5: Interaction with KRaider and conversion of triples into OWL Axioms.

4.4 Problems and Limitations

One of the main limitations of KRaider is that, at the moment, it cannot handle blank nodes: KRaider just ignores the triples that contain them. These nodes are important in order to convert RDF triples into complex OWL class expression or properties.

For instance, the OWL class expression $\exists hasChild.Person$ corresponds to the following RDF triples:

```
_:x rdf:type owl:Restriction .
_:x owl:onProperty hasChild .
_:x owl:someValuesFrom Person .
```

where `_:x` is a blank node. These triples are ignored by KRaider. In the future we plan to make KRaider handle blank nodes.

After multiple executions, it could happen that triples about an entity were already been extracted. Therefore, in order to improve the performances, KRaider should exploit caching. This is also future work. Moreover, we plan to allow the user to define filters that should be used during extractions.

5 Evaluation

We evaluated KRaider used inside SRL-Frame by performing several knowledge fragment extraction tasks on three different entities and an increasing recursion depth. The tests were performed on GNU/Linux machine equipped with Intel Core i7-5500U CPU @ 2.40GHz with 6 extractor threads.

Table 1 reports the number of extracted axioms and the running time in seconds averaged over 5 executions of KRaider (inside SRL-Frame) for different entities and recursion depth settings.

6 Conclusions

In this paper we proposed KRaider, a system that extract triples from different SPARQL Endpoints. The extracted triples are then converted into OWL axioms

Table 1: KRaider's results concerning knowledge fragment extraction of three different entities with an increasing recursion depth. (`db:` is equivalent to `http://dbpedia.org/resource/`)

	Recursion Depth					
	0		1		2	
URI	**# Axioms**	**Time**	**# Axioms**	**Time**	**# Axioms**	**Time**
`db:Leonardo_da_Vinci`	48	55.268	2956	130.029	100520.6	1271.571
`db:Angela_Merkel`	41	47.306	2412.4	107.771	160465.8	721.854
`db:Nikola_Tesla`	43	47.628	2139	87.499	62700.8	680.563

by another component. All these systems are integrated into a framework called SRL-Frame, which is still under development. SRL-Frame is based on OSGi technologies, which allows the system to dynamically install and start new services, hence making the framework flexible to changes.

KRaider's code is available as git repository at `https://bitbucket.org/machinelearningunife/kraider/`, whereas the code of SRL-Frame is available at `https://bitbucket.org/machinelearningunife/srl-frame/`.

In addition to the directions for future work presented in subsection 4.4, we plan to develop new type of extractors that exploit the other Linked Data interfaces. In particular, in the immediate future, we plan to integrate SPARQL-LD [5] into SRL-Frame as a triple extractor service to be used by KRaider.

References

1. Beek, W., Rietveld, L., Bazoobandi, H.R., Wielemaker, J., Schlobach, S.: LOD Laundromat: a uniform way of publishing other people's dirty data. In: ISWC 2012. pp. 213–228. Springer (2014)
2. Bizer, C., Heath, T., Berners-Lee, T.: Linked data: The story so far. In: Semantic services, interoperability and web applications: emerging concepts, pp. 205–227. IGI Global (2011)
3. Bühmann, L., Lehmann, J., Westpha, P.: DL-Learner – a framework for inductive learning on the semantic web. J. Web Semant. **39**, 15–24 (2016)
4. Cota, G., Riguzzi, F., Zese, R., Bellodi, E., Lamma, E.: A modular inference system for probabilistic description logics. In: Ciucci, D., Pasi, G., Vantaggi, B. (eds.) SUM 2018. LNCS, vol. 11142, pp. 78–92. Springer, Heidelberg, Germany (2018). https://doi.org/10.1007/978-3-030-00461-3_6, `http://mcs.unife.it/~friguzzi/Papers/CotRigZes-SUM18.pdf`
5. Fafalios, P., Yannakis, T., Tzitzikas, Y.: Querying the web of data with sparql-ld. In: Fuhr, N., Kovács, L., Risse, T., Nejdl, W. (eds.) Research and Advanced Technology for Digital Libraries. pp. 175–187. Springer International Publishing, Cham (2016)
6. Grau, B.C., Horrocks, I., Parsia, B., Ruttenberg, A., Schneider, M.: OWL 2 web ontology language mapping to RDF graphs (second edition) (12 2012)
7. Harth, A., Speiser, S.: On completeness classes for query evaluation on linked data. In: Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI-12) (2012)

8. Hartig, O.: An overview on execution strategies for linked data queries. Datenbank-Spektrum **13**(2), 89–99 (2013)
9. Hartig, O., Bizer, C., Freytag, J.C.: Executing sparql queries over the web of linked data. In: The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings. pp. 293–309. Springer (2009)
10. Hartig, O., Buil-Aranda, C.: Bindings-restricted triple pattern fragments. In: Debruyne, C., Panetto, H., Meersman, R., Dillon, T., Kühn, e., O'Sullivan, D., Ardagna, C.A. (eds.) On the Move to Meaningful Internet Systems: OTM 2016 Conferences. pp. 762–779. Springer International Publishing, Cham (2016)
11. Hellmann, S., Lehmann, J., Auer, S.: Learning of owl class descriptions on very large knowledge bases. International Journal on Semantic Web and Information Systems (IJSWIS) **5**(2), 25–48 (2009)
12. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - a large-scale, multilingual knowledge base extracted from Wikipedia. Semant. Web **6**(2), 167–195 (2015)
13. Montoya, G., Aebeloe, C., Hose, K.: Towards efficient query processing over heterogeneous rdf interfaces. In: Proceedings of the 2nd Workshop on Decentralizing the Semantic Web co-located with the 17th International Semantic Web Conference (ISWC 2018) (2018)
14. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. In: The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006. Proceedings. pp. 30–43. Springer Berlin Heidelberg (2006)
15. Riguzzi, F., Lamma, E., Bellodi, E., Zese, R.: BUNDLE: A reasoner for probabilistic ontologies. In: Faber, W., Lembo, D. (eds.) RR 2013. LNCS, vol. 7994, pp. 183–197. Springer Berlin Heidelberg (2013). https://doi.org/10.1007/978-3-642-39666-3_14
16. Shearer, R., Motik, B., Horrocks, I.: HermiT: A highly-efficient OWL reasoner. In: Dolbear, C., Ruttenberg, A., Sattler, U. (eds.) Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions, collocated with the 7th International Semantic Web Conference (ISWC-2008), Karlsruhe, Germany, October 26-27, 2008. CEUR-WS, vol. 432. CEUR-WS.org (2008)
17. Sirin, E., Parsia, B., Cuenca-Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. J. Web Semant. **5**(2), 51–53 (2007)
18. Stadler, C., Lehmann, J., Höffner, K., Auer, S.: LinkedGeoData: A core for a web of spatial open data. Semant. Web **3**(4), 333–354 (2012), `http://jens-lehmann.org/files/2012/linkedgeodata2.pdf`
19. Umbrich, J., Hose, K., Karnstedt, M., Harth, A., Polleres, A.: Comparing data summaries for processing live queries over linked data. World Wide Web **14**(5-6), 495–544 (2011)
20. Verborgh, R., Sande, M.V., Hartig, O., Herwegen, J.V., Vocht, L.D., Meester, B.D., Haesendonck, G., Colpaert, P.: Triple pattern fragments: A low-cost knowledge graph interface for the web. J. Web Semant. **37-38**, 184 – 206 (2016). https://doi.org/https://doi.org/10.1016/j.websem.2016.03.003, `http://www.sciencedirect.com/science/article/pii/S1570826816000214`
21. Vrandečić, D., Krötzsch, M.: Wikidata: A free collaborative knowledge base. Commun. ACM **57**, 78–85 (2014), `http://cacm.acm.org/magazines/2014/10/178785-wikidata/fulltext`