Avestia Publishing
Journal of Machine Intelligence and Data Science (JMIDS)
Volume 2, Year 2021
ISSN: 2564-3282
DOI: 10.11159/jmids.2021.006

Natural Language to SPARQL Query Builder for Semantic Web Applications

Neli Zlatareva, Devansh Amin
Department of Computer Science
Central Connecticut State University
1615 Stanley Street, New Britain, CT 06050, USA
zlatareva@ccsu.edu; devansh.amin@my.ccsu.edu

Abstract - *SPARQL is a powerful query language for an ever-growing number of Semantic Web applications. Using it, however, requires familiarity with the language which is not to be expected from the general web user. This drawback has led to the development of Question-Answering (QA) systems that enable users to express their information needs in natural language. This paper presents a novel dependency-based framework for translating natural language queries into SPARQL queries, which is built on the idea of syntactic parsing. The translation process involves the following steps: extraction of the entities, extraction of the predicate, categorization of the query's type, resolution of lexical and semantic gaps between user query and domain ontology vocabulary, and finally construction of the SPARQL query. The proposed framework was tested on our closed-domain student advisory application intended to provide students with advice and recommendations about curriculum and scheduling matters. The advantage of our approach is that it requires neither any laborious feature engineering, nor complex model mapping of a query expressed in natural language to a SPARQL query template, and thus it can be easily adapted to a variety of applications*.

Keywords**:** Information Retrieval, Natural Language Processing, Semantic Web, SPARQL, Question-Answering Systems.

1. Introduction

The use of Linked Data technologies for building Semantic Web applications has grown exponentially in the last decade. These new technologies are radically changing current web services by allowing users to post personalized queries directly to digital libraries of various information resources and databases. A huge number of these resources are interconnected via the *Linked Open Data Cloud* (https://www.lod-cloud.net/) and provide users with direct access via SPARQL endpoints to thousands of RDF/RDFS datasets. The bottleneck of this technology is that users must be familiar with the query language *SPARQL* (https://www.w3.org/TR/sparql11-query/) to place a query. Even the most user-friendly SPARQL endpoints, such as *DBPedia* (https://dbpedia.org/sparql), suggest some basic knowledge on writing SPARQL queries. More sophisticated queries, or federated queries over multiple SPARQL endpoints, require in-depth familiarity with SPARQL which is not to be expected from the users of Semantic Web applications. This difficulty is well recognized in the Semantic Web community and various techniques were suggested to address it. These techniques can be divided into two categories: information extraction and semantic parsing. The former aim to identify the main entities of the user's query and map them to ontology relations, most commonly by using pre-defined or automatically generated templates [1, 2, 3]. Semantic parsing techniques extract the meaning of the query by converting it into a syntactic structure [4, 5]. The main difficulty in this process is recognising the user's intention expressed in the natural language query so that it can be adequately translated into a SPARQL query.

This article presents a novel technique for converting user queries stated in natural language into SPARQL templates by dividing the translation process into sub-tasks that can be independently handled and

Date Received: 2021-09-28
Date Accepted: 2021-10-08
Date Published: 2021-11-10

processed by means of rule-based algorithms. The proposed technique was tested on a prototype Semantic Web application intended to provide students in our department with information and advice about programs, courses, faculty, etc. with the ultimate goal to serve as a recommender system for student advising and course registration. Currently, all this information is scattered among multiple websites and may require extensive browsing to collect it. Furthermore, the integration and interpretation of collected information is up to the student and it may take additional coordination with a faculty advisor to obtain the desired query result.

The article is structured as follows. In Section 2, we briefly discuss the RDF data model to provide some background for readers not familiar with the Semantic Web. The basic functionality of the student advisory application which was used to test the proposed technique is outlined in Section 3. Section 4 introduces the SPARQL query builder, and each one of the five subtasks involved in the translation process is discussed in detail and illustrated with examples. We conclude with a brief statement of our future plans.

2. RDF Data Model and Semantic Web Technologies

By utilizing Semantic web technologies, we can build applications which functionality goes beyond the traditional web-based search that intelligent assistants such as Siri, Cortana, or Google Assistant are intended to perform. This, however, may require building a new web infrastructure by converting needed information into a machine understandable format known as *Resource Description Framework* (RDF) (http://www.w3.org/RDF). RDF is a universal data model, which basic building block is the *triple*, a statement defining a relation between two web resources such as "Jones teaches Math101". Here *Jones* and *Math101* are called the *subject* and the *object* of the triple and *teaches* is the *predicate* expressing the relation between them. Each element of the triple <subject, predicate, object> is identified by its unique *International Resource Identifier* (IRI) (except for the object, which can also be a literal of any XML datatype) thus making it globally accessible across the web. Problem domains are described as sets of triples which represent directed graphs. Independent graphs can be easily merged via common nodes (subjects or objects of triples) thus forming a linked data network. Processing linked data is based on the following *linked data principles* [7]:

1. Use unique identifiers, IRIs, to name every entity (resource), physical or abstract, that exists in the world.
2. Use HTTP IRIs to allow users to look up those resources to gather information about them.
3. When looking up an IRI, use RDF-based representation and SPARQL query language to access that IRI.
4. Include links to associated IRIs to allow for automatic discovery of related data.

Implementation of these principles allows for effortless management of large quantities of linked data and facilitates their integration and processing. The later is performed by the *SPARQL Protocol and RDF Query Language* (SPARQL) (http://www.w3.org/TR/rdf-sparql-query/). SPARQL is a very rich language offering four different query forms depending on the expected result. The SELECT query uses a pattern matching algorithm to retrieve specific information, while DESCRIBE and CONSTRUCT queries return RDF graphs that can be combined with other graphs for further processing. ASK query is similar to SELECT query in that it also utilizes pattern matching, but it answers whether there is at least one match or no match at all.

RDF provides the data model, but the *Web Ontology Language (OWL)* (http://www.w3.org/TR/owl-features/) is the current W3C recommendation supporting the development of Semantic Web applications. Building such applications is similar to the development of Knowledge-Based Systems. The knowledge base, called here the *ontology*, is defined as "… an explicit, formal specification of a shared conceptualization" [7]. The most notable difference between Semantic Web ontologies and traditional knowledge bases is that knowledge bases typically reflect the view of a domain expert, or a group of experts, while ontologies tend to reflect the consensus view of the community expressed by precisely defined terms. Ontologies also resemble relational database models, but in addition to relations between data they implicitly define formal rules of inference which allow data processing to be carried out by automated reasoners. This in turn, expends the scope of services provided by Semantic Web applications to allow for personalized response to a broad range of user queries.
Semantic Web reasoners are based on the so-called *Description Logics* (DLs) [8]. These are decidable fragments of first-order logic intended to achieve favourable trade-offs between expressivity and scalability. Between themselves, DLs defer by the set of

constructors they utilize to represent data and perform reasoning. One of the most expressive DLs, called *SROIQ(D)*, underlines the latest version of OWL, *OWL 2*. It defines an ontology as a triple *<T-Box, A-Box, R-Box>*, where:

- The *T-Box* defines domain terminology expressed as a hierarchy of classes related by subsumption, $C \sqsubseteq D$, and equivalence, $C \equiv D$, relations. It also includes a disjunction constructor, a special class expression Self: $\exists S.Self$, and allows for qualified number restrictions $\geq n\ S.C$ and $\leq n\ S.C$ to express statements such as "a course with at least/at most 6 graduate students".
- The *A-Box* defines the domain description stating class membership of individuals ($a \in C$), property relations between individuals (<a, R, b>), and equality relations between individuals ($a = b$).
- The *R-box* defines complex domain relations as combination of properties, $R_1 \circ R_2 \sqsubseteq S$, allowing statements such as "hasTakenCS151 $\circ$ hasTakenCS152 $\sqsubseteq$ canTakeCS153", as well as describes properties of properties such as inverse properties, symmetry, reflexivity, irreflexively and disjunctiveness of properties.

The next section outlines a prototype application utilizing the described technologies, which was used to validate the proposed natural language to SPARQL query builder technique.

3. Student Advisory Application: design and functionality

The core of the student advisory application is a bot intended to provide students with enhanced experience when seeking information about programs, courses, faculty, etc. offered by our department. Domain knowledge utilized by the advisory bot integrates information from multiple websites such as University catalogue, course scheduling system, departmental website, and more. Currently, to find the needed information a student would browse these different websites and at the end may not be able to find the answer they were looking for not necessarily because the information was not there, but because it might not be easily accessible.

As stated in Section 2, Semantic web technologies allow to conventionalise integration of different information resources by "merging" them into a single linked data graph. To illustrate the functionality of the application, consider the following scenario. A transfer student is looking for courses to enrol given that she: 1.) has credit for CS 151, CS 152, and CS 253; 2.) can only take courses on Mondays and Wednesdays after 3 pm; 3.) is a full-time student but wants to take only 4 courses; 4.) prefers to take core courses rather than electives to ensure that she is on track to graduate in two years; and 5.) wants to make sure that all prerequisites for senior level courses are satisfied.

A single SPARQL query on a linked data network will be sufficient to address the example scenario. The problem, however, is that the student is not expected to know SPARQL. Instead, a user-friendly interface should be available to translate user query into one or more SPARQL queries retrieving the relevant information. The SPARQL query builder described in Section 4 is intended to address this task.

The architecture of the student advisory application is shown on Figure 1. The output from the SPARQL query builder is processed by a Java-based module which uses Apache Jena API (http://jena.apache.org) and Pellet reasoner [9] to acquire and integrate relevant domain information from the backend repository containing the domain ontology and the datasets.

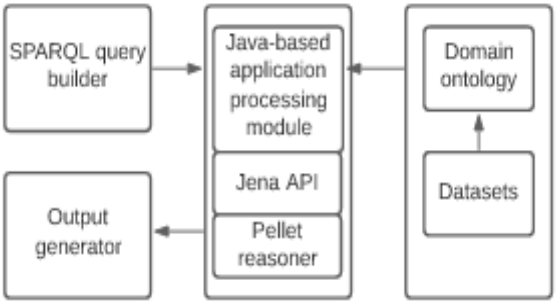

Figure 1. Student advisory application architecture

To build the domain ontology (the *T-Box*), we used Protégé (http://www.protege.stanford.edu), which currently is the best-known open-source ontology editor. The illustrate the scope of the underlying domain, we show the class hierarchy, object, and data properties hierarchies on Figures 2, 3, and 4, respectively.

The domain ontology is stored in a Turtle format which can be easily processed by the Java application

module. Datasets containing information from various web resources (the *A-Box*) are also stored as Turtle files.

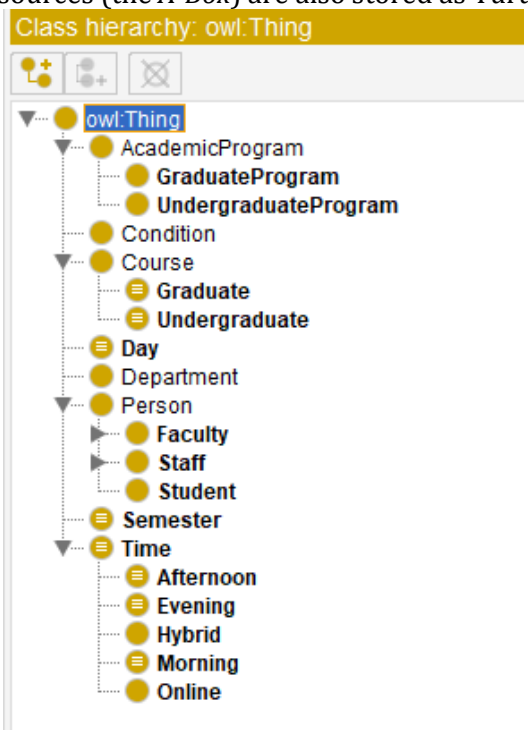

Figure 2. Class hierarchy

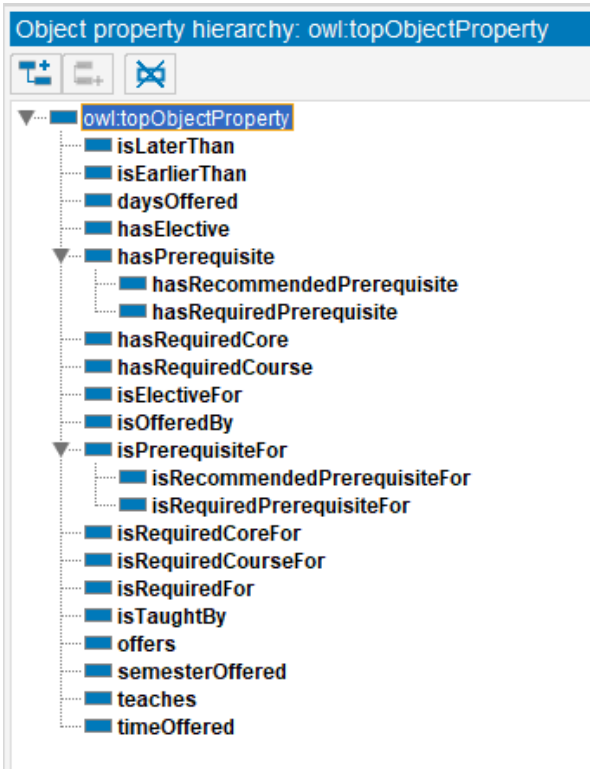

Figure 3. Object properties hierarchy

The ontology and the datasets are processed using Apache Jena API which provides extensive Java libraries for managing RDF datasets and OWL ontologies and allows for easy integration of SPARQL queries. In addition, Jena allows for integration of OWL 2 reasoner Pellet [9] by a plug-in called Openllet (http://github.com/Galigator/openllet). In our application, we use Pellet to extend the initial graph (the ontology and relevant datasets) before the SPARQL query is posted. The obtained result is returned as an HTML table.

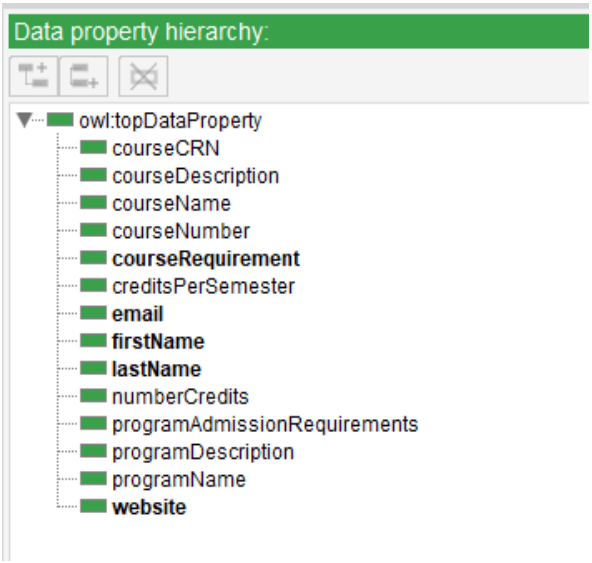

Figure 4. Data Properties Hierarchy

Next, we discuss the SPARQL query builder and the dependency parsing technique that it is built upon.

4. SPARQL Query Builder

4.1 Basic Notation and Terminology

The technique presented in this paper aims to build SPARQL queries from a natural language text. It is based on idea of *syntactic parsing* (or *dependency parsing*) which converts a sentence into a syntactic structure by building a dependency parse tree [10]. The later contains typed labels denoting the grammatical relationships for each word in the sentence. To carry out this process, we used spaCy [11], which is a Python/Cython library for advanced natural language processing. spaCy has a fast and accurate syntactic dependency parser and a rich API for navigating the dependency tree. For readers unfamiliar with spaCy, we want to clarify some of the terminology used further in the paper. The terms ***head*** and ***child*** are used to describe words connected by a single edge in the dependency tree

(https://spacy.io/usage/linguistic-features#navigating). The term ***dep*** denotes an edge label describing the type of syntactic relation between the child and the head nodes. The syntactic dependency scheme described below is adopted from ClearNLP [12].

In the generated parse tree, each *child* has only one *head,* but a head may have multiple children. The *head* can be accessed by the *Token.head* attribute and its children can be accessed by the *Token.children* attribute. *Token.lefts* and *Token.rights* attributes return sequences of syntactic children that occur before and after the *Token*. *Token.subtree* attribute is used to get the whole phrase by its syntactic head, and it returns an ordered sequence of tokens. Two data structures, *stack* and *visited,* are initialized to empty Python lists. These are used to store the tokens while traversing the dependency tree.

4.2 SPARQL Query Builder: Architecture and functionality

The architecture of the SPARQL query builder presented in this paper is shown on Figure 5.

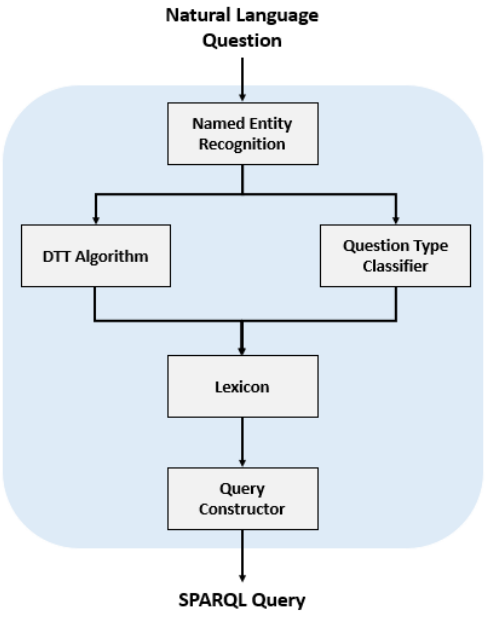

Figure 5: Query builder architecture

It is intended to support the following types of queries:

- **Single fact query.** These are over a single RDF triple <subject, predicate, object>. The query result is either the subject or the object of the triple. Example shown on Figure 6.

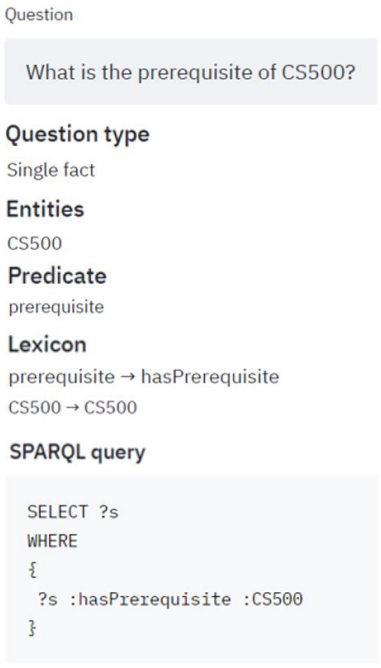

Figure 6: User interface example of single fact query

The processing of this query type is shown on Figure 7.

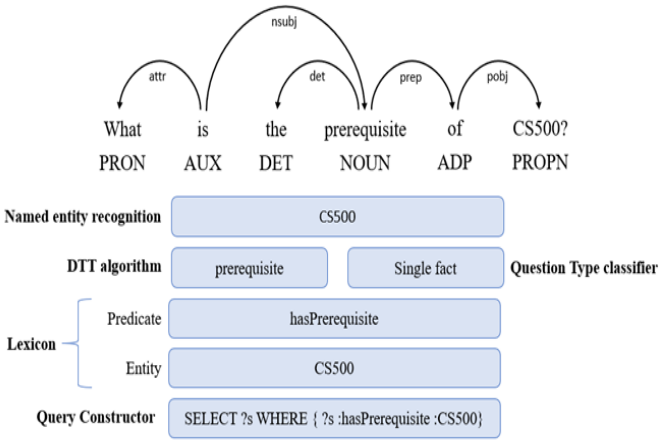

Figure 7: Example of a single-fact query

- **Single fact with type query.** The template for this query identifies the type in a single triple. Example shown on Figure 8.

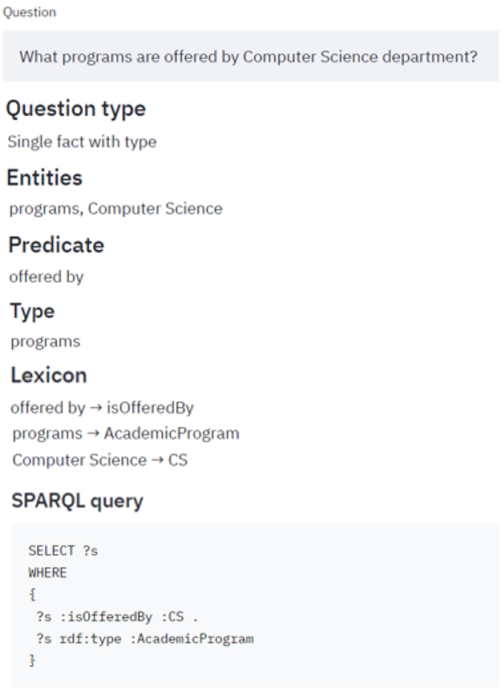

Figure 8: User interface example of single fact with type query

The processing of this query is shown on Figure 9.

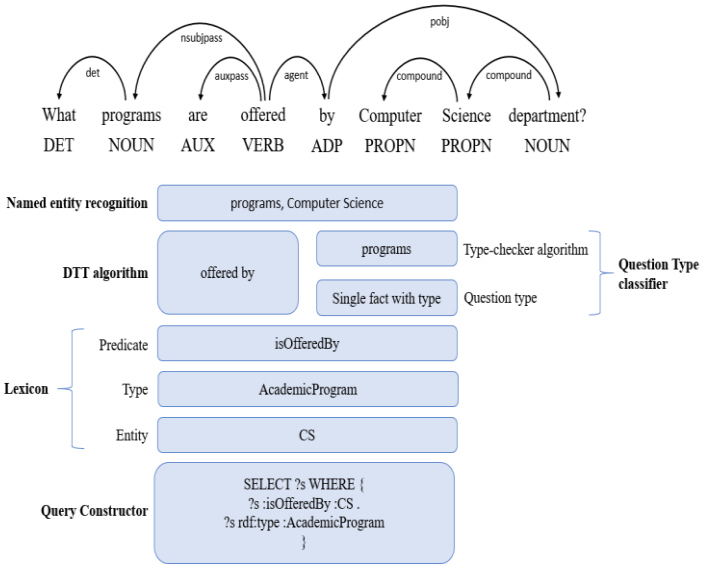

Figure 9: Example of a single fact with type query

- **ASK queries**. These queries expect a true / false answer. Example shown on Figure 10.

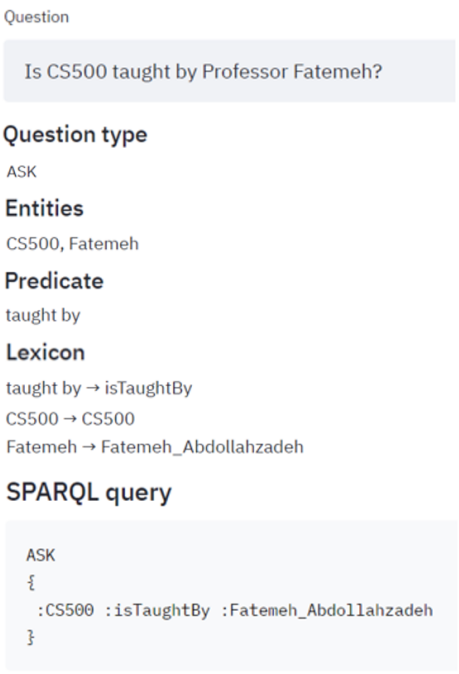

Figure 10: User interface example of ASK query

The processing of this query is shown on Figure 11.

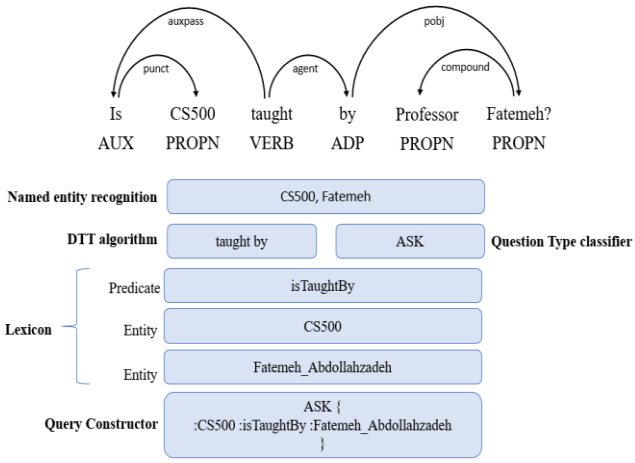

Figure 11: Example of an ASK query

Query builder modules are discussed next.

4.2.1 Named Entity Recognition

Named Entity Recognition (also known as entity identification) is a subtask of information extraction that seeks to locate and classify atomic elements in a text into predefined categories such as person names, locations, organizations and more. This step is essential for

gathering the entities that serve as input to processing algorithms.

4.2.2 Dependency Tree Traversal Algorithm

The Dependency Tree Traversal (DTT) algorithm extracts the predicate by traversing the dependency tree generated from the user query. The pseudocode of the algorithm is shown on Figures 12.1, 12.2, and 12.3. The algorithm requires the following pre-processing steps:

Step 1: Remove spaces and punctuations from the end of the question.

Step 2: Use regular expressions to check if the question contains open and closed parenthesis in which case remove the parentheses and the data inside them from the question and store it in a Python list. It is important to note that removing the parenthesis from the question does not change the dependency tree.

Step 3: Use regular expressions to check if the question contains any punctuations. We use a dictionary to keep track of the words with punctuations, where the *key* is the word without the punctuation and the *value* is the word with the punctuation. If a word containing an apostrophe S ('s), then the apostrophe S is not removed from the user's question because removing it results in a change of the dependency tree. The rest of the punctuations are extracted and removed from the user question.

The DTT algorithm first identifies the root node of the tree. The root node is a node with no incoming edges. Usually, the root token will be the main verb of the sentence (although this may not be true for unusual sentence structures, such as sentences without a verb). The root node is identified by iterating over the tokens and selecting the token which head is the same as the token itself i.e., *token.head = token*.

```
Algorithm 1: DTT Algorithm
   Input  : A user's question and entities.
   Output: A predicate.
1  left, right = filter(root)
2  if notEmpty(left, right) then
3  |  setFlag()
4  else
5  |  appendToStack(root)
6  end
7  for children in (left, right) do
8  |  traverseAndAppendToStack(root)
9  end
```

Figure 12.1 Dependency tree traversal algorithm

Next, we explore the left and right children of the root. When we have a "single fact" or "single fact with type" queries, the left and right children will contain the predicate of the former and the predicate or the type of the latter. Since we do not remove stop words from the user's query, the left and right children can contain such words. These are the most common words in any natural language sentence, namely the, is, in, for, etc. A function *filter* is created to filter out each child of the left and right children whose subtree contains words that are in stop words and the entity. The function *filter* returns two lists. If we have a non-empty list for the left and right children, then the *root* is part of the predicate, and *flag* is set to *true*. If we have an empty list for either left or right children, then we add the *root* to *stack* and *visited*. The latter are initially set to empty Python lists and are used to store tokens while traversing the dependency tree. We iterate over each child in the left children and right children and add it to *stack,* if the child contains a word that is not in the stop words and the entity. If the *flag* is equal to *true*, the root word is added to *stack* and *visited*. The *flag* is now set to *false* to avoid having the root word added twice to *stack* and *visited*.

```
Algorithm 1: Algorithm 1 continued
10 while notEmpty(stack) do
11 |  pointer = topOfStack()
12 |  if pointer == root then
13 |  |  popFromStack()
14 |  |  if isStopWord(pointer) then
15 |  |  |  continue
16 |  |  else
17 |  |  |  appendToPredicate(pointer)
18 |  |  end
19 |  else if isVisited(pointer) then
20 |  |  element = popFromStack()
21 |  |  if containsPunctuation(element) then
22 |  |  |  element = addPunctuation(element)
23 |  |  end
24 |  |  appendToPredicate(element)
25 |  else
26 |  |  appendToVisited(pointer)
27 |  |  if notEmpty(pointer.children) then
28 |  |  |  traverseAndAppendToStack(children)
29 |  |  end
30 |  end
31 end
```

Figure 12.2 Dependency tree traversal algorithm

The algorithm now iterates over the *stack* until it is empty. We get the element that is on the top of the

stack and assign it to a variable *pointer* to check the following conditions:

Condition 1: If the *pointer* is equal to the root, pop an element from *stack* and *visited.* If *pointer* is not in stop words, append the element to *predicate;* else return the control to the beginning of the loop.

Condition 2: If *pointer* is in *visited*, pop an element from *stack* and *visited.* If the element had punctuation attached to it before pre-processing, add the punctuation back to *element* and append *element* to *predicate*.

Condition 3: If the above conditions fail, add *pointer* to *visited*. If *pointer* contains children, iterate over the left and right children of the *pointer*. If children exist, iterate over each child of the children; if the child contains a word that is not in stop words and entity, add the child to *stack*.

Algorithm 1: Algorithm 1 continued

```
32 if notEmpty(paren_mid) then
33 |  addParenthesis()
34 end
35 predicate = cleanAndSort(predicate)
36 if notEmpty(paren_end) then
37 |  addParenthesis()
38 end
39 return predicate
```

Figure 12.3 Dependency tree traversal algorithm

We can now add the punctuations removed during pre-processing back to the predicate. We then remove any stop words from the beginning and the end of *predicate* and sort *predicate* in the right order as they appear in the user question. Finally, the predicate is converted from the list to a string by joining it by space.

4.2.3 Question Type Classifier

The Question Type Classifier uses a rule-based algorithm to classify the question to a type of the SPARQL query. As stated above, our framework currently supports the following three types of queries:

- Single fact. If the question contains only a single entity, then it is classified as a single fact (see example on Figure 6).
- Single fact with type. If the question contains more than one entity, a rule-based algorithm *type-checker* (pseudocode shown on Figure 13) checks whether the question contains a type. Extracting the type from the user query, is carried out depending on the category of the question. We distinguish between (i) questions starting with Wh (i.e., what, when, where, who, whom, which, whose and why), and (ii) all others. For each category, we have defined dependency rules to extract the type.
- ASK. If the question contains more than one entity and does not have a type, then the question is of type ASK.

Algorithm 2: Type-checker Algorithm

Input : A user's question and entities.
Output: The type.

```
1  document = nlp(question)
2  for token in document do
       // dep_ means dependency
3      if question.startswith("wh") then
4          if token.dep_ in ("nsubj","nsubjpass") then
5              type_head = token
6              break
7          end
8      else
9          if token.dep_ == "dobj" then
10             type_head = token
11             break
12         else if token.dep_ == "pobj" and
             previousToken.dep_ == "det" then
13             type_head = token
14     end
15 end
16 return entityContaining(type_head)
```

Figure 13: The type-checker algorithm

4.2.4 Lexicon

The role of the lexicon is to map the vocabularies (properties and entities) used in the user query to those from the application ontology. There might be inconsistencies between the two which we refer to as a *lexical gap* and a *semantic gap*. The former defines to the difference between query and ontology vocabularies, while the later refers to the difference between expressed information needs and the adopted ontology representation. The proposed lexicon component is intended to overcome both gaps. We used Sentence-BERT (SBERT) [13] to compute the sentence embeddings of all the properties and entities in the ontology and saved the embeddings as a *PyTorch Tensor* (https://pytorch.org/docs/stable/tensors.html). SBERT is a modification of the pre-trained BERT network that *uses Siamese and Triplet networks* [13] to derive semantically meaningful sentence embeddings that can be compared using cosine-similarity. We run the SBERT

model with different pooling strategies like MEAN, MAX, and CLS, out of which MEAN pooling strategy worked well for our semantic textual similarity (STS) [14] task. We used cosine similarity as the similarity function. Using the SBERT model, we compute the sentence embeddings of the predicate and entities extracted from the user query and perform a semantic comparison with all the property embeddings for the former, and entity embeddings for the latter using cosine similarity. We sort the similarity scores from the highest to the lowest and select the top 5 similar labels. Next, we compute the Jaccard similarity coefficient of the label (predicate or entity) with the label having the highest cosine similarity (most similar label). The Jaccard coefficient (https://en.wikipedia.org/wiki/Jaccard_index) measures similarity between finite sample sets and is defined as the size of the intersection divided by the size of the union of the sample sets. If the Jaccard similarity is greater than the 0.7 thresholds, we assign the most similar label to the label, else we check if the label is found in the aliases of the similar labels.

```
Function Lexicon(question, label, group)
    Data:
        question: A user's question.
        label: A property or entity label.
        group: property or entity.
    Result:
        A label.
    similar_labels = cosineSimilarity(label, group)
    most_similar_label = similar_labels[0]
    jc_score = jaccardSimilarity(label, most_similar_label)
    if (jc_score > 0.7) then
        label = most_similar_label
    else
        for similar_label in similar_labels do
            aliases = getAlias(similar_label, group)
            if aliasesContains(label) then
                label = similar_label
                break
            end
        end
        else
            label = cosineSimilarity(question, similar_labels)
        end
    end
    return label
end
```

Figure 14: Lexicon function

If both conditions fail, we use SBERT to get the embeddings for the user question and do a cosine similarity with the similar labels and select the label with the highest similarity score and return the label. This process is repeated for all the entities. Pseudocode of the Lexicon function is shown on Figure 14.

4.2.5 Query Constructor

This module uses the information provided by Lexicon (predicate and entities) and Question Type Classifier modules to build the SPARQL query. Each question type has its own SPARQL template. The role of Query Constructor is to build the SPARQL query and to return the SPARQL query results to the user.

5. Conclusion

This article presents a novel technique for translating natural language queries into SPARQL queries. The framework implementing it, the SPARQL query builder, uses a dependency rule-based algorithm to convert user queries to "user" triples. These are validated by the lexicon and further converted into RDF triples to construct a SPARQL query that fetches the answers from the underlying ontology via a JAVA-based application processing module. The advantage of the presented technique is that it requires neither any laborious feature engineering, nor does it require any complex model mapping of a natural language question to a query template and then to a SPARQL query. Since the dependency tree traversal and the type-checker algorithms do not require any domain specific knowledge the proposed framework can be applied to arbitrary domains.

In our future work, we plan to add additional functionality to support complex SPARQL queries and evaluate the system on open-domain datasets such as LC-QuAD (http://lc-quad.sda.tech/lcquad1.0.html).

References

[1] Dimitrakis E., K. Sgontzos, M. Mountantonakis, and Y. Tzitzikas - Enabling Efficient Question Answering over Hundreds of Linked Datasets. *Post-proceedings of the 13th International Workshop on Information Search, Integration, and Personalization (ISIP'2019*), 2019.

[2] Abujabal A., M. Yahya, M. Riedewald, and G. Weikum. - Automated template generation for question answering over knowledge graphs. *Proceedings of the 26th international conference on world wide web*. International World Wide Web Conferences Steering Committee, 2017.

[3] Diefenbach D., K. Singh, and P. Maret. -WDAqua-core1: A Question Answering service for RDF Knowledge Bases. *WWW'18: Companion Proceedings of the The Web Conference*, 2018.

[4] Shaik S., P. Kanakam, S. Hussain,and D. Suryanarayana - Transforming Natural Language Query to SPARQL for Semantic Information Retrieval, International Journal of Engineering Trends and Technology (IJETT), Volume-41 Number-7, 2016.

[5] [Online] Available: Semantic Parsing Natural Language into SPARQL: Improving Target Language Representation with Neural Attention (groundai.com)

[6] Berners-Lee T. Linked Data - Design Issues. [Online] Available: http://www.w3.org/DesignIssues/LinkedData.html, 2006.

[7] Gruber, T. – A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition, 5(2), pp. 199-220.

[8] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. (editors) *The Description Logic Handbook. Theory, implementation, and applications*. 2010, Cambridge University Press.

[9] Sirin E., Parsia B., Cuenca Grau B., Kalyanpur A., Katz Y. Pellet: A Practical OWL-DL Reasoner, http://www.cs.ox.ac.uk/people/bernardo.cuencagrau/publications/PelletDemo.pdf

[10] Honnibal M, Johnson M. - An improved non-monotonic transition system for dependency parsing. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015:1373–1378.

[11] "spacy.io," 2016. [Online]. Available: https://spacy.io

[12] Choi J. and M. Palmer - Guidelines for the CLEAR Style Constituent to Dependency Conversion, 2012.

[13] Reimers, N. and I. Gurevych - Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. 3973-3983. 10.18653/v1/D19-1410. 2019.

[14] Agirre E., D. Cer, M. Diab, A. Gonzalez-Agirre, and Weiwei Guo. *sem 2013 shared task: Semantic textual similarity, including a pilot on typed similarity. In **SEM 2013: The Second Joint Conference on Lexical and Computational Semantics*, 2013, Association for Computational Linguistics