Proceedings of the 7th World Congress on Electrical Engineering and Computer Systems and Sciences (EECSS'21)
~~Prague, Czech Republic~~ Virtual Conference - July 2021
Paper No. CIST 108
DOI: 10.11159/cist21.108

Processing Natural Language Queries in Semantic Web Applications

Neli Zlatareva, Devansh Amin
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 based on the idea of syntactic parsing. The translation process involves the following five 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 vocabularies; and 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 domains.

***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 user intention expressed in a natural language query so that it can adequately be translated into a SPARQL query. The framework presented in this paper addresses this issue by dividing the translation process into five sub-tasks that can be independently handled and processed by means of rule-based algorithms.

Our research came to address a practical need as we were developing a pilot Semantic Web application intended to help students with information and advice regarding program transfer, course registration, schedules, and more at our university. Currently, this information is scattered among multiple databases and may require extensive browsing to collect it. Furthermore, the integration and interpretation of collected information is up to the user and it may take additional problem solving to arrive at the solution of the intended query. 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, requires building a new web infrastructure by converting needed information into a machine understandable RDF format. More about the pilot application can be found in [6]. The emphasis of this paper is on building a user-friendly interface allowing users to interact with Semantic Web applications like ours in natural language.

The paper is organized as follows. In Section 2, we briefly review the RDF data model to provide some background for readers not familiar with the Semantic Web. Section 3 describes the proposed framework for building a natural language interface to SPARQL endpoints. We conclude with a brief statement of our future plans.

2. RDF Data Model and Semantic Web Technologies

Semantic Web is built on a universal data model called the Resource Description Framework (RDF) (http://www.w3.org/RDF). It was proposed by Tim Berners-Lee [7] and is rooted in the following linked data principles:

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

Implementation of these principles allows for effortless management of large quantities of web data and facilitates their integration and processing.

RDF provides the data model, but the Web Ontology Language (OWL) (http://www.w3.org/TR/owl-features/) and the SPARQL Protocol and RDF Query Language (SPARQL) (http://www.w3.org/TR/rdf-sparql-query/) are the current W3C recommendations supporting the development of Semantic Web applications.

The basic building block of the RDF is the triple, a statement defining the 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 relationship between them. Each element of the triple <subject, predicate, object> is identified by its unique URI (except for the object, which can be a literal of any XML datatype) thus making it globally accessible across the web.

Data represented in this format is stored in the so-called triple stores. A triple store is in fact a knowledge base expressed in a chosen representation language, typically OWL, and it consists of the following 3 parts:

- Domain terminology, the so-called *T-Box,* which is expressed as a hierarchy of classes. It is formally described by subsumption and equivalence relationships between classes, $C \sqsubseteq D$ and $C \equiv D$, respectively. The latest version of OWL, OWL 2, includes also 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".
- Domain description, the so-called *A-Box,* containing facts about the specific domain. It defines class membership of individuals ($a \in C$), property relations between individuals ($<a, R, b>$), and equality relations between individuals ($a = b$).
- Domain relations, the so-called *R-Box* defines complex properties such as inverse properties, symmetry, reflexivity, irreflexively and disjunctiveness of properties, as well as combination of properties $R_1 \circ R_2 \sqsubseteq S$, allowing statements such as "hasTakenCS151 $\circ$ hasTakenCS152 $\sqsubseteq$ canTakeCS153".

The prototype version of our student advisory application which was used to test the proposed NL interface is implemented as an OWL-based triple store containing information from multiple university databases intended to provide students with personalized advice and recommendations about their course of study. Examples to illustrate the intended functionality are shown in [6]. All queries are posted in SPARQL. This paper addresses the need to allow users to post queries in natural language thus making such applications available to the general user.

3. Building a Natural Language Interface to a SPARQL endpoint

3.1 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 [8]. The later contains typed labels denoting the grammatical relationships for each word in the sentence. To carry out this process, we used spaCy [9], 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 [10].

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 nodes while traversing the dependency tree.

3.2 SPARQL Query Builder: Architecture and functionality

The architecture of the SPARQL query builder presented in this paper is shown on Figure 1.

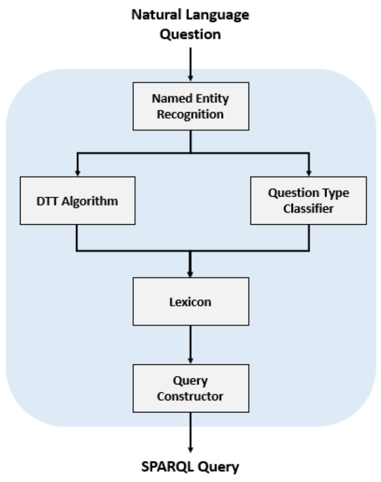


Figure 1: Query builder architecture.

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. For example: “What is the prerequisite of CS500?”. The processing of this query type is shown on Figure 2.
- **Single fact with type query.** The template for this query identifies the type in a single triple. For example: "What programs are offered by Computer Science department?". The processing of this query is shown on Figure 3.
- **ASK queries**. These queries expect a true / false answer. For example: “Is CS500 taught by Professor Fatemeh?”. (see Figure 4).

Query builder modules are discussed next.

3.2.1 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.

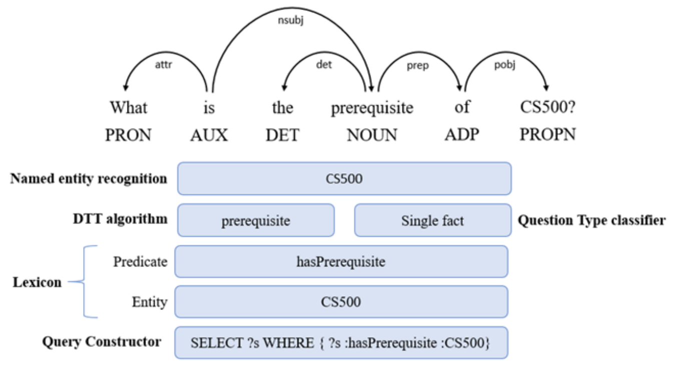


Figure 2: Example of a single-fact query

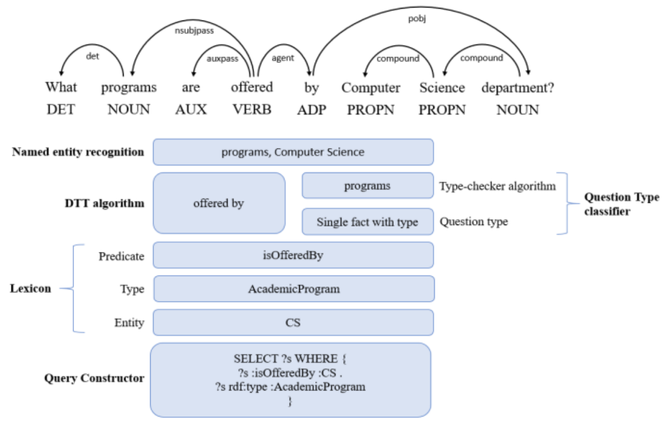


Figure 3: Example of a single fact with type query

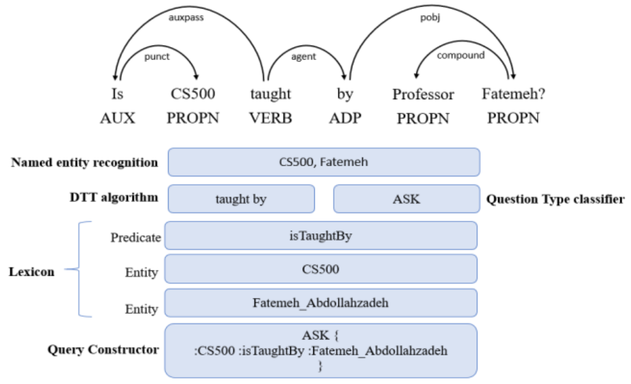


Figure 4: Example of an ASK query

3.2.2 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 5.1, 5.2, and 5.3. 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.
$left, right = filter(root)$
if $notEmpty(left, right)$ then
    $setFlag()$
else
    $appendToStack(root)$
end
for $children$ in $(left, right)$ do
    $traverseAndAppendToStack(root)$
end
```

Figure 5.1 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[da1]. 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
while $notEmpty(stack)$ do
    $pointer = topOfStack()$
    if $pointer == root$ then
        $popFromStack()$
        if $isStopWord(pointer)$ then
            continue
        else
            $appendToPredicate(pointer)$
        end
    else if $isVisited(pointer)$ then
        $element = popFromStack()$
        if $containsPunctuation(element)$ then
            $element = addPunctuation(element)$
        end
        $appendToPredicate(element)$
    else
        $appendToVisited(pointer)$
        if $notEmpty(pointer.children)$ then
            $traverseAndAppendToStack(children)$
        end
    end
end
```

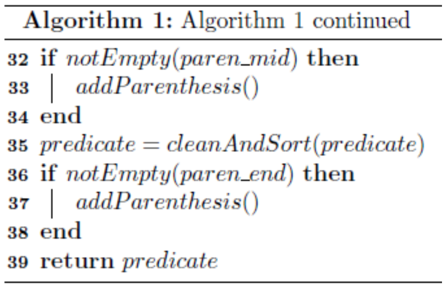
Figure 5.2 Dependency tree traversal algorithm

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
if $notEmpty(paren\_mid)$ then
    $addParenthesis()$
end
$predicate = cleanAndSort(predicate)$
if $notEmpty(paren\_end)$ then
    $addParenthesis()$
end
return $predicate$
```

Figure 5.3 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.

3.2.3 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 Figure 2 for an example).
- Single fact with type. If the question contains more than one entity, a rule-based algorithm *type-checker* (pseudocode shown on Figure 6) 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.

```
Algorithm 2: Type-checker Algorithm
  Input  : A user's question and entities.
  Output: The type.
document = nlp(question)
for token in document do
    // dep_ means dependency
    if question.startswith("wh") then
        if token.dep_ in ("nsubj", "nsubjpass") then
            type_head = token
            break
        end
    else
        if token.dep_ == "dobj" then
            type_head = token
            break
        else if token.dep_ == "pobj" and
         previousToken.dep_ == "det" then
            type_head = token
    end
end
return entityContaining(type_head)
```

Figure 6: The type-checker algorithm

- ASK. If the question contains more than one entity and does not have a type, then the question is of type ASK.

3.2.4 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) [11] 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 [11] 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) [12] 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. If both the 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 7.

```
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 7: Lexicon function

3.2.5 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.

4. Conclusion

This paper presents a novel approach to translating NL queries into SPARQL queries. The proposed framework 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. The advantage of our framework is that it requires neither any laborious feature engineering, nor does it require any complex models 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] Zlatareva N. – A Specialized Recommender Agent for the Semantic Web. *Proc. of the 4th World Congress on Electrical Engineering and Computer Systems and Sciences (EECSS'18)*, August 2018, Madrid, Spain.

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

[8] 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.

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

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

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

[12] 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