PAROT: Translating Natural Language to SPARQL

Peter Ochieng

Taita Taveta university,
Nairobi,Kenya
onexpeters@gmail.com

Abstract

This paper provides a dependency based framework for converting natural language to SPARQL. We present a tool known as PAROT (which echos answers from ontologies) which is able to handle user's queries that contain compound sentences, negation, scalar adjectives and numbered list. PAROT employs a number of dependency based heuristics to convert user's queries to user's triples. The user's triples are then processed by the lexicon into ontology triples. It is these ontology triples that are used to construct SPARQL queries. From the experiments conducted, PAROT provides state of the art results.

Keywords: SPARQL, Natural Language Processing, Ontologies, Query

1. Introduction

In our bid to develop an ontology based chatbot, we envision developing a tool that would allow users to use their natural language (NL) and have a near natural conversation with a tool which fetches facts (answers) contained in an ontology based knowledge base (KB). This requires us to employ a plugin tool that translates the user's statements written in NL to SPARQL query language (SPARQL Working Group, 2013), a W3C recommended language for querying ontologies. We experimented with various NL to SPARQL tools such as Aqua-Log (Lopez et al., 2005), CASIA@12 (He et al., 2014), Querix (Kaufmann et al., 2006), AutoSPARQL (Lehmann & Bühmann, 2011), K-Extractor (Tatu et al.,

[1]Since 1880.

2016), SPARK (Ferré, 2017) that currently exist in literature in order to select the best tool. The best tool was to be selected based on its precision and recall value (i.e. its ability to fetch correct and all require answers). However, we realized that despite the tools converting a number of user's queries to the correct SPARQL queries, the tools' precision and recall values drastically dropped for queries which contained:

1. Opposing scalar adjectives such as in the query *which is the **longest** and **shortest** river that traverses Mississippi ?*, (Zhao et al., 2017) estimates that 12% of the total errors in queries generated by their gAnswer tool is due to the fact that it does not support superlatives and comparatives in its implementation. This underscores the importance of handling scalar adjectives.
2. Negation such as *which rivers do **not** flow through Alaska ? or which river **neither** flows through Alaska nor Mississippi ?*.
3. Numbered list such as *list **five** rivers that flow through Alaska ?*
4. Compound sentences *which female actor played in Casablanca and is married to a writer born in Rome ?* . (Zhao et al., 2017) estimates that 9% of the total errors in queries generated by their gAnswer tool is due to the fact that it does not handle queries with unions or filters.

In addition to these weaknesses, most of the state of art tools use techniques that are not able to capture the entire vocabulary of the underlying knowledge base i.e. they don't generalize the entire knowledge base adequately. This affects the word disambiguation process hence reducing their precision and recall values.

This research addresses the above mentioned key challenges by introducing the following key concepts:

1. Design a lexicon that that is able:
    - To fully represent the vocabulary of the underlying knowledge base therefore helping in resolving word ambiguities that exist in the user's query.

    - To tag adjective entities in the knowledge base with their positive and negative scalars. Through this we are able to resolve the problem of opposing scalar adjectives when converting NL to SPARQL.

2. We develop a number of high coverage syntactic heuristics which can convert different scenarios of possible questions to correct SPARQL queries.

From the evaluation, the developed technique outperforms gAnswer (Zhao et al., 2017), which was the top performing tool in QALD-9 challenge (Usbeck et al., 2018a). This is due its ability to effectively disambiguate words and its high coverage of user questions.

2. Literature Review

In this section, we review state of the art applications that converts NL to SPARQL as well as those that convert NL to SQL. We highlight the key techniques used by a tool and discuss whether it can handle the challenges discussed in section 1. AquaLog (Lopez et al., 2005) is an ontology independent question answering system for the Semantic Web. It is composed of a linguistic component to map user query to query triples. It is these query triples that are further processed into an ontology compliant triples from where answers are derived. The linguistic component is composed of the GATE infrastructure (Cunningham et al., 2001) and resources to annotate the user query. The annotations in the user query include verbs, nouns, tokens etc. The component also employs JAPE grammars which expand annotations embedded by the GATE by identifying terms, relations, question indicators (which/who/when, etc.) and patterns or types of questions. AquaLog does not contain components to deal with scalar adjectives, numbered list and compound sentences. CASIA@12 (He et al., 2014) is a question answering system over linked data. After generating a number of possible phrase to semantic item mappings, it then uses Markov logic network (MLN) for disambiguation and finally form a SPARQL query. CASIA@12 does not handle scalar adjectives, negation, numbered list and compound sentences. DEANNA (Yahya et al., 2012) also translates a question in NL into a structured query. The key element of DEANNA is the use integer linear program (ILP) to solve the disambiguation of terms to semantic items. Querix (Kaufmann et al., 2006) is a pattern matching ontology independent natural language interface (NLI). Querix uses the Stanford parser to syntactically analyze the input query. From the syntax tree the query analyzer extracts the sequence of the key word categories such as Noun (N), Verb (V), Preposition (P), Wh-Word (Q), and Conjunction (C). Based on the generated word categories a query skeleton is generated. WordNet is used to supply all synonyms to the verbs and nouns in the query. It then matches the skeleton with triples in the ontology . In Querix ambiguities are not resolved automatically rather users are asked for clarifications in a pop-up dialog menu window to disambiguate. Queries has a disadvantage that it only allows users to write queries starting with which, what, how many, how much, give me or does hence cannot handle questions starting with terms such as "List". It also does not handle negation, scalar adjectives and extensively relies on WordNet which makes query generation process slow. PANTO (Wang et al., 2007) utilizes Stanford parser (Klein & Manning, 2003) to generate a parse tree from the user submitted query. It then extracts nominal phrase constituents in the parse trees . The nominal phrases in the parse trees are extracted as pairs to form an inter-mediate representation called QueryTriples. It the utilizes the knowledge in the ontology, to map QueryTriples to OntoTriples which are represented with entities in the ontology. Finally, together with targets and modifiers extracted from the parse trees, OntoTriples are interpreted as SPARQL. PANTO can handle conjunctions / disjunctions, negation, comparatives and superlatives. It however cannot handle opposing scalar adjectives and numbered list. AutoSPARQL (Lehmann & Bühmann, 2011) uses supervised machine learning to generate a SPARQL query based on positive i.e. resources which should be in the result set of the SPARQL query, and negative examples, i.e. resources which should not be in the result set of the query. The user can either start with a question as in other QA systems or by directly searching for a relevant resource. He or she can then select an appropriate result, which becomes the first positive example. After that, he is asked a series of questions on whether a resource should also be contained in the result set. These questions are answered by a yes or a no. This feedback allows the supervised learning method to gradually learn which query to generate. AutoSPARQL faces the challenge of portability to a different Knowledge Base (KB) (Sander et al., 2014). The effort of learning the positive and negative examples also increases drastically with the size of the KB (Sander et al., 2014). SPARK (Ferré, 2017) is another tool for processing NL keyword to SPARQL. Its output is a ranked list of SPARQL queries. Its key steps include: term mapping, construction of the query graph and query ranking. Ranking of query applies a probabilistic model based on the Bayesian Theorem. Its key challenge involves choosing an option out of the ranked query list since this requires an expert in SPARQL who has knowledge on the underlying KB (Sander et al., 2014). Exploiting the recent success of deep learning, a number of studies introduce deep learning neural network based method to convert NL to structured query languages. Research in (Hao et al., 2017) applies a bidirectional long short term memory(LSTM) (Hochreiter, Sepp and Schmidhuber, 1997) network to convert NL to SPARQL. Bidirectional LSTM is employed to capture the context of a word in relation to both the words before and after it. Their technique exploits the top key words in the submitted user question to extract candidate answers from the knowledge base. The words are then linked to the correct answer tokens by learning their relatedness. WDAqua (Usbeck et al., 2018b) generates SPARQL query from natural language by employing rule-based combinatorial approach to generate leveraging the semantics encoded in the underlying knowledge base. Other state of the art tools for converting NL to SPARQL include TeBaQA(Usbeck et al., 2018b), Elon(Usbeck et al., 2018b) and QASystem(Usbeck et al., 2018b). A complete review of NL to SPARQL tools is discussed in (Bouziane et al., 2015) , (Cimiano & Bielefeld, 2011) and (Diefenbach et al., 2018).

To convert NL to SQL, research in (Iyer et al., 2017), employs a bidirectional LSTM to generate the best SQL query from a given NL. It applies encoder-decoder model proposed in (Ahmad & Hunt, 2015). In the decoder, the conditional probability distribution of the SQL token is predicted based on the previous combination of SQL token embeddings. They also incorporates human feedback to improve the learning process. Other studies that have exploited neural networks to convert NL to structured language include (Cai et al., 2018),(Yu et al., 2018) and (Gur et al., 2018). One major challenge with Neural Network based method is that they try to represent the whole knowledge base using a few training examples. This becomes a challenge when the model meets new vocabulary that it had not seen in the training data hence affecting the prediction of neural network based. Table 1 gives a summary of an evaluation of selected state of the art tools on whether they can handle negation, opposing scalar adjectives, compound sentences and the key disambiguation technique a tool applies.

Table 1: Evaluation of selected NL to structured language tools

Tool	**Structured Language**	**Negation**	**Opposing Scalar adjectives**	**Compound sentences**	**Disambiguation Technique**
AquaLog (Lehmann & Bühmann, 2011)	SPARQL	No	No	No	GATE infrastructure
CASIA@12 (He et al., 2014)	SPARQL	No	No	No	Markov Logic Network
DEANNA (Stoilos et al., 2005)	SPARQL	No	No	Yes	Integer linear programming
Querix (Kaufmann et al., 2006)	SPARQL	No	No	No	User
PANTO (Wang et al., 2007)	SPARQL	Yes	No	Yes	Query patterns
AutoSPARQL (Lehmann & Bühmann, 2011)	SPARQL	No	No	Yes	Machine learning
FREyA (Damljanovic et al., 2011)	SPARQL	No	No	Yes	User
QuestIO	SPARQL	No	No	Yes	User
K-Extractor (Tatu et al., 2016)	SPARQL	No	No	Yes	Ranking
SPARK (Ferré, 2017)	SPARQL	No	No	Yes	Ranking
SQLnet (Xu et al., 2017)	SQL	yes	No	Yes	Neural Networt(LSTM)
DiaSQL (Gur et al., 2018)	SQL	yes	No	Yes	Neural Networt(LSTM)
SyntaxSQLNet(Yu et al., 2018)	SQL	yes	No	Yes	Neural Networt(LSTM)
PAROT	SPARQL	yes	yes	Yes	Syctactic heuristic and Lexicon technique

3. PAROT Architecture

3.1. Step 1: Identifying targets.

Given a user query submitted in natural language, the first task is to identify targets words from the query. A target (or projection) word is a variable that will be placed directly after SELECT key word in a SPARQL query. To help in identifying target words in a user submitted query, we use a typed dependency parser such as Stanford typed dependency parser (Marneffe & Manning, 2015). The dependency parser provides a simple description of grammatical relationships that exists between the words in the user submitted query. To extract the target words in the parsed query, we categorize the queries into two categories i.e.

- The Wh (WRB, WP, WDT) queries.
- The non WH queries.

3.1.1. Targets in Wh based queries

This category is composed of queries which start with *Wh* (i.e. *what, when, where, who, whom, which, whose, why, and how).* To identify target words in this category of queries, we apply two key set of rules in equation 1 and 2.

$$\forall w_x, w_y.(nsubj(w_x, w_y) \implies Target(w_y)) \tag{1a}$$

$$\forall w_x, w_y, w_z.(nsubj(w_x, w_y) \wedge conj(w_y, w_z) \implies Target(w_y) \wedge Target(w_z)) \tag{1b}$$

$$\forall w_x, w_y.(nsubjpass(w_x, w_y) \implies Target(w_y)) \tag{2a}$$

$$\forall w_x, w_y, w_z.(nsubjpass(w_x, w_y) \wedge conj(w_y, w_z) \implies Target(w_y) \wedge Target(w_z)) \tag{2b}$$

Here, $dep(x, y)$ is a dependency that exists between words x and y. The words of a sentence compose the constants of a domain over which the functions operate. The rules in (1a) and (1b) apply in a non relation query (see section 3.2.1 for definition of a relational and non-relational query). They basically identify the nominal subjects in the user submitted query and flags them as targets. Equation (1a) applies for a query where there is no conjunct relation between the head subject of the query and any other nominal. For example, when a user submits a query such as *What is the area of the most populated state ?*, using Stanford dependency parser, the dependency diagram in figure 1 is generated. The grounded version of formula (1a) is shown below.

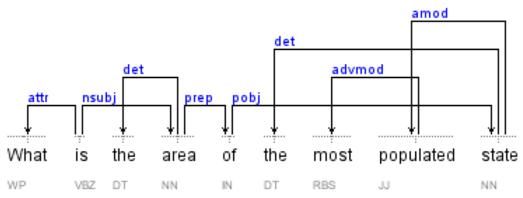

Figure 1: Showing dependency

$$nsubj(is, area) \implies Target(area)$$

The dependency $nsubj(is, area)$ holds between the words *is* and *area*. Therefore, the nominal *area* is selected as the target of the query. If a user submits a query such as *What is the area and population of the most populated state ?*, Stanford dependency viewer generates dependency shown in figure 2. Since in

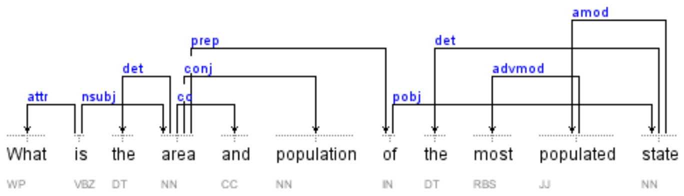

Figure 2: Showing dependency

this query the head subject *area* is connected to another nominal by a coordinating conjunction *"and"* , the formula in equation (1b) is applied.

$$nsubj(is, area) \wedge conj(area, population) \implies Target(area) \wedge Target(population)$$

Both the nominals *area* and *population* are selected as targets. This rule also captures scenarios where more than one nominal is connected to the head subject by a coordinating conjunction, such as *and*, *or* and "," such as *What is the population, area and capital of the most populated state?*

The rules in equation (2a) and (2b) are applicable in a relational based query. They flag subjects word in a query by identifying the passive nominal subjects in the user submitted query. Equation (2a) is applicable where the head subject is not connected to any other nominal via a conjunction. For example in the query *Which German actor was killed in a road crash ?*, the nominal *actor* is selected as the target word.

$$nsubjpass(killed, actor) \implies Target(actor)$$

The rule in equation (2b) captures scenarios where one or more nominals are connected to the head subject by a coordinating conjunction, such as *and*, *or* and "," such as in the query *Which German actor and musician were killed in a road crash?*. The nominals actor and musician are picked as target words.

$$\begin{aligned} nsubjpass(killed, actor) \wedge conj(actor, musician) \implies & Target(actor) \wedge \\ & Target(musician) \end{aligned}$$

3.1.2. Targets in non-WH queries

To flag out target words in a non-Wh queries, we use the functions in equation 3 and 4. Equation (3a) and (3b) applies in a *non-Wh* query that the direct object is not connected to a preposition such as *Give me all the rivers that traverse Mississippi*(see dependency diagram in figure 3). The rule in equation (3b) applies where the direct object is connected to one or more nouns via a conjuction (e.g. *Give me all rivers and lakes that traverse Mississippi*)

$$\forall w_x, w_y.(dobj(w_x, w_y) \implies Target(w_y)) \tag{3a}$$

$$\forall w_x, w_y, w_z.(dobj(w_x, w_y \wedge conj(w_y, w_z) \implies Target(w_y) \wedge Target(w_z)) \tag{3b}$$

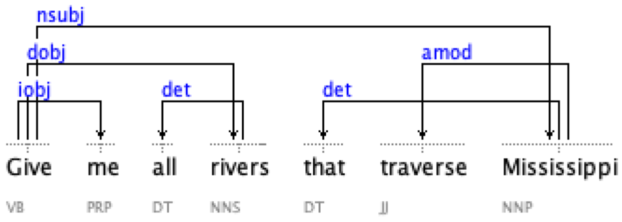

Figure 3: Showing dependency

$$\forall w_x, w_y.(pobj(w_x, w_y) \implies Target(w_y)) \tag{4a}$$

$$\forall w_x, w_y, w_z.(pobj(w_x, w_y \wedge conj(w_y, w_z) \implies Target(w_y) \wedge Target(w_z)) \tag{4b}$$

Equation (4a) and (4b) applies in a non-Wh query that the head of a noun phrase folows a preposition. Equation (4a) applies in query such as *In which country does the Nile start?*. Here, the head noun *country* is not connected to any noun via a conjuction. Equation (4b) applies to a query where the head noun is connected to one or more nouns via a conjuction such as in the query *In which country and continent does the Nile start?*. The head noun must be connected to a preposition.

3.2. Step 2: Identifying user triple pattern

SPARQL query is composed of a set of triple patterns known as graphs patterns. The graphs patterns are placed directly after the WHERE key word or after the target variables in the SPARQL query. The triple patterns are of the form of $< subject >< predicate >< object >$ where the subject, predicate and object may be variables (SPARQL Working Group, 2013). The idea therefore in this section is to process a user submitted query to identify potential triples that will be used to construct the SPARQL graphs. Triples identified from the user query are referred here as user triples. To identify user triples from the submitted query, we categorize it into either:

1. Relational phrase based query.
2. Non-relational phrase based query.

3.2.1. Identifying user triple pattern in a relation based user query

Relation based user query is a query which contains at least a relational phrase linking two nominals. A relation phrase can be a transitive verb (e.g. in the query *"which rivers traverse Alaska?"*, *traverse* is a relational phrase) or intransitive verb followed with prepositional complement (e.g. *"which river flows through Alaska?"*, *flows through* is a relational phrase). Therefore, a relational phrase linking two nominals may be a verb, a verb followed directly by a preposition or a verb followed by nouns, adjectives, or adverbs ending in a preposition (A. Fader, S. Soderland, 2011).

To identify user triples in a relation based query, we apply Algorithm 1. The algorithm accepts a user submitted query (sentence) which is composed of a number of words as its input. It then evaluates if a sentence is compound using the function *checkCompound(S).* The function $checkCompound(S)$ applies a number of syntactic constraints to categorize a sentence as compound or not. The syntactic constraints are shown in figure 4. A compound sentence should be composed of the following sequence:

1. A verb followed by a preposition followed by a noun, conjuction and a verb (e.g. *Which female actor* ***played in Casablanca and is married*** *to writer born in Rome*)
2. verb followed by a noun followed by a conjuction and a noun(e.g. *Which river* ***traverses Mississippi or Alaska***)

Algorithm 1 Extracting user triples from a relation based user query

```
    Input Sentence $S=(w_1, w_2 \cdots w_n)$.
    Output $UserTriples$.
1: Given a sentence $S = \{w_1, w_2, ..., w_n\}$
2: Check if $S$ is compound i.e $CheckCompound(S)$. (equation 5)).
3: if $CheckCompound(S)$=true. then
4:     $Break(S) = (s_1(cc)s_2)$
5: else
6:     $UserTriple = GenerateTriple(S)$.
7: end if
8: return $UserTriples$
```

3. Noun followed by a conjuction followed by a noun and a verb(e.g. *Which **rivers and lakes traverse** Alaska*
4. An adverb, superlative followed by a conjuction followed by an adverb, superlative followed by a verb(e.g. *Which is the **least and most populated state** in America*)

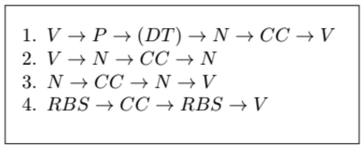
1. $V \rightarrow P \rightarrow (DT) \rightarrow N \rightarrow CC \rightarrow V$
2. $V \rightarrow N \rightarrow CC \rightarrow N$
3. $N \rightarrow CC \rightarrow N \rightarrow V$
4. $RBS \rightarrow CC \rightarrow RBS \rightarrow V$

Figure 4: POS tag patterns to flag compound sentences in a relation based queries

Using Stanford dependency parser, we translate the syntanctic constrains in figure 4 into a set of formulas shown in equation 5.

$$\forall w_e, w_h, w_i, w_j, w_k.(prep(w_e, w_h) \wedge pobj(w_h, w_i) \wedge cc(w_e, w_j) \wedge conj(w_e, w_k) \implies Compound(w_a, w_b, \cdots, w_n)) \tag{5a}$$

$$\forall w_e, w_h, w_i, w_j.(dobj(w_e, w_h) \wedge cc(w_h, w_i) \wedge conj(w_h, w_j) \implies Compound(w_a, w_b, \cdots, w_n)) \tag{5b}$$

$$\forall w_e, w_h, w_i, w_j.(nsubj(w_e, w_h) \wedge cc(w_h, w_i) \wedge conj(w_h, w_j) \implies Compound(w_a, w_b, \cdots, w_n) \tag{5c}$$

$$\forall w_e, w_h, w_i, w_j.(advmod(w_e, w_h) \wedge cc(w_h, w_i) \wedge conj(w_h, w_j) \implies Compound(w_a, w_b, \cdots, w_n) \tag{5d}$$

Consider the sentence, *S=which rivers traverse Mississipi or Alaska*. Applying rule (5b), the sentence is categorized as compound as shown below.

$$dobj(traverse, Mississippi) \wedge cc(Mississippi, or) \wedge$$
$$conj(Mississippi, Alaska) \implies Compound(S)$$

A sentence S that is categorized as compound has to broken into two simple sentences (s_1 and s_2). The sentences are joined by a conjuction that was linking them in the user query i.e. ($s_1(cc)s_2$). The function $Break(S)$ which is iterative applies rule in equation 6 to extract two simple sentences from the compound sentence S.

$$Break(S) \equiv s_1(cc)s_2 \tag{6}$$

where

$$s_1 = w_a, w_b, \cdots, w_i \; s_2 = w_a, w_b, \cdots, w_{e-1}, w_{j+1} \cdots w_n \tag{7a}$$

$$s_1 = w_a, w_b, \cdots, w_h \; s_2 = w_a, w_b, \cdots, w_{h-1}, w_j \tag{7b}$$

$$s_1 = w_a, \cdots, w_h, \cdots, w_{j+1} \cdots w_n \; s_2 = w_a, w_{i+1}, \cdots, w_j, \cdots, w_n \tag{7c}$$

$$s_1 = w_a, \cdots, w_h, w_{j+1} \cdots w_n \; s_2 = w_a, \cdots, w_{h-1}, w_j \cdots, w_n \tag{7d}$$

The rule in equation (7a) applies to a compound sentence identified by the rule in equation (5a). Likewise, (7b) applies to (5b) ,(7c) to (5c) and (7d) applies to

(5d). Here, we assume the last word of a sentence is w_n. Applying rule (7b), the compound sentence: *S=which rivers traverse Mississipi and Alaska* is broken down into two simple sentences i.e.

s_1= *Which rivers traverse Mississippi.*

s_2= *Which rivers traverse Alaska.*

$cc = and$

Finally, the algorithm identifies triples through the function *GenerateTriples.* For each simple sentence s_i, *GenerateTriple* function, identifies user triples in it by extracting two subsequent head nouns and connects them using a relational phrase that links them. For instance, in s_1 above we have the nouns *rivers* and *Mississippi* as two subsequent head nouns and *traverse* is the relational phrase linking them, therefore, *GenerateTriple* will generate a single triple from s_1 i.e. {rivers *traverse* Mississippi}. Likewise, in s_2 a single user triple {rivers *traverse* Alaska} will be generated. Once the user triples have been established, we generate all triple arrangements that predict possible ways in which concepts in the user triples may be modelled in the underlying ontology. For example, the user triple {*rivers traverse Mississippi*} predicts that the undelying ontology for instance has modelled the concepts *river* and *Mississippi* as { *river* : $flows\_through$ *Mississippi*}. However, the undelying ontology may have modellled the concepts as { *Mississippi :hasRiver river*}. To capture both these possibilities, for each user triple we extract, we create a second one where the concepts in the subject and object position are interchanged. Therefore, for the user triple {*rivers traverse Mississippi*} we create another {*Misssippi traverse river*} where the concepts *Mississippi* and *rivers* are interchanged. In this example we generate the following user triples. {*river traverse Mississippi OR Mississippi traverse river*} {*river traverse Alaska OR Alaska traverse river*}. The correct arrangement of concepts as modelled in the ontology will be resolved by the lexicon.

Consider the user query *Which female actor played in Casablanca and is married to a writer born in Rome ?*, based on the rule in equation (5a), the query is marked as a compound sentence. The query is therefore broken into two simple sentences i.e.

s_1= *Which female actor played in Casablanca*

s_2=*Which female actor is married to a writer born in Rome.*

cc = *And*

When *GenerateTriple* function is applied to both s_1 and s_2, the following user triples are generated.

$GenerateTriple(s_1)$= {*actor played_in Casablanca*}

$GenerateTriple(s_2)$={*actor married_to writer, writer born_in Rome*}

The user triples are then expanded to predict possible positions in the undelying ontologies. {*(actor played_in Casablanca, Casablanca played_in actor)* }

{*(actor married_to writer, writer married_to actor)(writer born_in Rome, Rome born_in writer)*}

The final user triples generated from the query is shown in Listing 1.

Listing 1: Sample user triples

```
{
(actor played_in Casablanca) OR (Casablanca played_in actor)
                                        AND
(actor married_to writer) OR (writer married_to actor)
(writer born_in Rome) or (Rome born_in writer)
}
```

The correct position of the concepts as modelled in the undelying ontology will be resolved by the lexicon as discussed in section 3.5. There are two special cases where *GenerateTriple* applies the rules in equation 8. The rules basically apply in scenarios where it is not explicit which nouns a verb is linking. The rules are applied;

1. When a query starts with *Who* e.g *Who killed Ceasar ?*
2. When a query contains a verb that does not lie between any two nouns (

i.e. when a verb is at the end of a sentence such as in the query *In which continent does the Nile traverse ?*.

Rule (8a) applies for the first scenario while (8b) for the second.

$$\forall w_e, w_h, w_i.(nsubj(w_e, w_h) \wedge dobj(w_e, w_i) \implies Triple(?w_h, w_e, w_i)) \vee Triple(w_i, w_e, ?w_h)) \tag{8a}$$

$$\forall w_e, w_h, w_i, w_j.(pobj(w_e, w_h) \wedge prep(w_j, w_e) \wedge nsubj(w_j, w_i) \implies Triple(w_i, w_j, w_h) \vee Triple(w_h, w_j, w_i) \tag{8b}$$

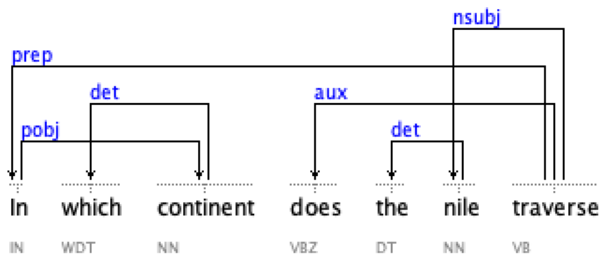

Figure 5: Showing dependency

Consider the query *Who killed Ceasar ?*, applying the rule in equation (8a),

$$nsubj(killed, who) \wedge dobj(killed, Ceasar) \implies Triple(?who, killed, Ceasar) \vee Triple(Ceaser, killed, ?who)$$

the user triple {*?who killed Ceasar*} or {*Ceaser killed ?Who*} is generated. Again consider the user query *In which continent does the Nile traverse ?*. The dependency diagram is shown in figure 5. Applying the rule in equation (8b),

$$pobj(In, continent) \wedge prep(traverse, In) \wedge nsubj(traverse, Nile) \implies Triple(Nile, traverse, continent) \vee Triple(Continent, traverse, Nile)$$

Listing 2: Sample user triples

```
{
```

```
(nile traverse continent) or (continent traverse nile)

}
```

3.2.2. Identifying user query triple pattern in non-relation based user query

A non-relational query is a query (sentence) which has no relational phrase linking any of its nominals. For instance, the sentence *What is the area of the most populated state?* is a non-relation based query. To identify triples that exist in this category of queries, we use Algorithm 2.

Algorithm 2 Triple extraction Algorithm in a non-relational query

```
   Input $SentenceS = (w_1, w_2 \cdots w_n)$.
   Output $UserTriples$.
1: Given a sentence $S = (\{w_1, w_2, ..., w_n\})$
2: Check if it is compound i.e $CheckCompound2$(S) (equation 9,10 or 11)).
3: if $CheckCompound2(S) == true$ then
4:     $(s_1, s_2) = Break2(S)$
5: else
6:     $UserTriples = GenerateTriple2$(S).
7: end if
8: return $UserTriples$
```

The function *CheckCompound*2 establishes whether a non-relation query is compound or not based on two key rules developed. The rules in equation 9 and 10 applies for *Wh* and *non-Wh* based queries respectively. The rule in equation 11 applies for a a non relational query that contains the pattern $JSS \rightarrow CC \rightarrow JJS \rightarrow N$ i.e. an adjective, superlative followed by a conjuction followed by an adjective, superlative followed by a noun(e.g. *Which is the* ***longest and shortest river*** *in America*).

$$\begin{aligned} &\forall w_d, w_f, w_g, w_k.(nsubj(w_d, w_f) \wedge cc(w_f, w_g) \wedge conj(w_f, w_k) \\ &\implies Compound(w_a, w_b, \cdots, w_n) \end{aligned} \tag{9}$$

$$\begin{aligned}&\forall w_d, w_f, w_g, w_k.(dobj(w_d, w_f) \wedge cc(w_f, w_g) \wedge conj(w_f, w_k)\\ &\implies Compound(w_a, w_b, \cdots, w_n)\end{aligned} \tag{10}$$

$$\begin{aligned}&\forall w_e, w_h, w_i, w_j.(amod(w_e, w_h) \wedge cc(w_h, w_i) \wedge conj(w_h, w_j)\\ &\implies Compound(w_a, w_b, \cdots, w_n)\end{aligned} \tag{11}$$

A compound sentence is broken into two simple sentences, s_1, s_2 as shown in the rule in equation 12. where

$$Break2(S) \equiv s_1(cc)s_2 \tag{12}$$

Where in a compound sentence identified by rule 9 and 10

$s_1 = w_a, w_b, \cdots, w_f, w_{k+1}, \cdots, w_n$

$s_2 = w_a, w_b, \cdots, w_{f-1}, w_{g+1} \cdots w_n$

while in a compound sentence identified by rule 11

$s_1 = w_a, \cdots, w_h, w_{j+1} \cdots w_n$

$s_2 = w_a, \cdots, w_{h-1}, w_j \cdots, w_n$

Consider the user query S=*What is the population and area of the most populated state ?*. The dependency diagram is shown in figure 2. Applying equation 10,

$$\begin{aligned}&nsubj(is, population) \wedge cc(population, and) \wedge conj(population, area)\\ &\implies Compound(S)\end{aligned}$$

Since the sentence is compound, the $Break2(S)$ function is applied to break it into two simple sentences s_1 and s_2 i.e.

s_1=*What is the population of the most populated state*

cc= And

S_2=*What is the area of the most populated state*

Finally, user triples are identified through the function $GenerateTriples2$. To identify triples, $GenerateTriples2$ exploits the presence of;

1. Preposition (*father of Tom or mountain in Germany*).
2. Genitive's construction (Tom's father).

The preposition *"of"* signals that a given noun posess a specified property e.g. in the query *What is the* ***area of*** *the most populated state* suggests that the noun *state* has a property *area.* The preposition *in* is a signal that a given object belong to a noun e.g. *What is the highest* ***mountain in*** *Germany* depicts that Germany has an object of the type *mountain.* Therefore based on the type of preposition used, $GenerateTriples2$ uses two key set of rules to extract triples from a user submitted query. When using a preposition to extract triples, the general syntactic constraint in non-relation based query is $N \rightarrow IN \rightarrow (A^*) \rightarrow N$ i.e. a noun followed by a preposition followed by a noun. Sometimes a determinant and an adjective may exists as depicted by A^*. The rules in equations 13 apply for a query where *of* preposition is used. Equation (13a) and (13b) are exploited for *Wh* and *non-Wh* based queries respectively.

$$\forall w_u, w_x, w_y, w_z.(nsubj(w_u, w_x) \wedge prep(w_x, w_y) \wedge pobj(w_y, w_z) \wedge (w_y = \text{“}of\text{”}) \implies Triple(w_z, (w_x\_w_y), ?k)) \tag{13a}$$

$$\forall w_u, w_x, w_y, w_z.(dobj(w_u, w_x) \wedge prep(w_x, w_y) \wedge pobj(w_y, w_z) \wedge (w_y = \text{“}of\text{”}) \implies Triple(w_z, (w_x\_w_y), ?k)) \tag{13b}$$

Hence, applying $GenerateTriples2$ (rule (13a)) to s_1 and s_2

$$(nsubj(is, population) \wedge prep(population, of) \wedge pobj(of, state) \wedge (of = \text{“}of\text{”}) \implies Triple(state, (population\_of), ?k))$$

$$(nsubj(is, area) \wedge prep(area, of) \wedge pobj(of, state) \wedge (of = \text{“}of\text{”}) \implies Triple(state, (area\_of), ?k))$$

$GenerateTriples2(s_1)$=$\{State\ population\_of\ \ ?x\}$

$GenerateTriples2(s_2)$=$\{State\ area\_of\ \ ?x\}$

$UserTriples$=$\{State\ population\_of\ \ ?x\ AND\ State\ area\_of\ \ ?y\}$

Listing 3: User triples

```
{
State population_of ?x
         AND
State area_of ?y
}
```

For a case where the *in* preposition is used, equation (14a) and (14b) applies for *Wh* and *non-Wh* based questions respectively.

$$\forall w_u, w_x, w_y, w_z.(nsubj(w_u, w_x) \land prep(w_x, w_y) \land pobj(w_y, w_z) \land (w_y = \text{``}in\text{''}) \implies Triple(w_z, ?k, w_x)) \lor Triple(w_x, ?k, w_z)) \tag{14a}$$

$$\forall w_u, w_x, w_y, w_z.(dobj(w_u, w_x) \land prep(w_x, w_y) \land pobj(w_y, w_z) \land (in = \text{``}in\text{''}) \implies Triple(w_z, ?k, w_x) \lor Triple(w_x, ?k, w_z)) \tag{14b}$$

Consider the user query *Which is the highest mountain in Germany ?*, applying the rule in equation (15a),

$$nsubj(is, mountain) \land prep(mountain, in) \land pobj(in, Germany) \land (w_y = \text{``}in\text{''}) \implies Triple(Germany, ?k, mountain) \lor Triple(mountain, ?k, Germany))$$

The user triple{*Germany ?k Mountain*} *or* {*Mountain ?k Germany*} is extracted.

Here, we try to preempt all the possible the arrangements of concepts in the underlying ontology. The term *Germany* could be modeled in the ontology to occupy the subject position such as in the triple {*Germany :hasMountain Adelegg*} or it can be modeled to occupy the object position such as {*Adelegg :belongTo Germany*}. The exact triple to be selected will be determined by the lexicon. When it comes to genitive's complement the rule in equation 15 is applied. The syntactic constraint applied is $N \rightarrow POS \rightarrow N$ i.e. it involves two nouns, the head and the dependent (or modifier noun)(e.g German's flag).

The dependent noun modifies the head by expressing some property of it. The rule in (15a) is applicable in a *non-Wh* query while (15b) applies in a *Wh* based queries.

$$w_u, w_x, w_y, w_z.(dobj(w_u, w_x) \wedge poss(w_x, w_y) \wedge possessive(w_y, w_z) \\ \implies Triple(w_y, ?k, w_x)) \vee Triple(w_x, ?k, w_y)) \tag{15a}$$

$$w_u, w_x, w_y, w_z.(nsubj(w_u, w_x) \wedge poss(w_x, w_y) \wedge possessive(w_y, w_z) \\ \implies Triple(w_y, ?k, w_x)) \vee Triple(w_x, ?k, w_y)) \tag{15b}$$

Consider the query *What is Angela's birth name?*(see dependency in figure 6), applying the rule in equation (15b),

$$nsubj(is, name) \wedge poss(name, Markel) \wedge possessive(Markel,' s) \\ \implies Triple(Markel, ?k, name) \vee Triple(name, ?k, Markel))$$

The triple {*Markel, ?k name*}∨ {*name,?k, Markel*} is generated.

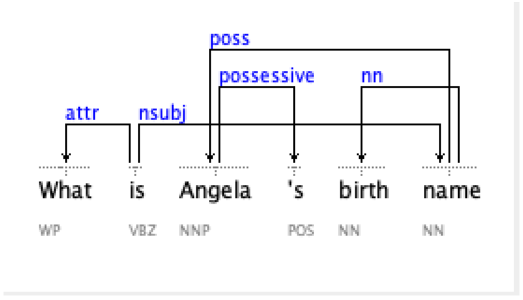

Figure 6: Dependency diagram

3.3. Step 3: Handling adjectives

To handle adjectives, we categorize them into two groups

1. Scalar adjectives.
2. Non-scalar adjectives.

3.3.1. Scalar adjectives

Scalar adjectives are those which communicate the idea of scale. In OWL ontologies, scalar adjectives can be mapped to an owl:DatatypeProperty. To flag out scalar adjectives, we use two two key syntactic constraints

$$\boxed{JSS \rightarrow NN | RBS \rightarrow VBN \rightarrow NN}$$

Figure 7: POS tag patterns for scalar adjective

The syntactic constraint requires that a scalar adjective matches any of the the POS tag pattern shown in figure 7. The pattern limits a scalar adjective to be

1. An adjective, superlative followed by a noun (e.g., longest river).
2. An adverb, superlative followed by a verb followed by a noun (e.g., most populated state)

A scalar adjective is a signal that a given noun posses a property indicated by the adjective. For example the combination *longest river* is an indication that the noun river has a property *length.* We therefore translate the syntactic contraints in figure 7 to the rules in equation 16 and 17. The rules are used to create triples from a query that contains a scalar adjective.

$$\forall w_u, w_x, w_y.(amod(w_x, w_u) \wedge (w_u \equiv JJS) \implies Triple(w_x, root(w_u), ?k)) \quad (16)$$

$$\forall w_u, w_x, w_y.(advmod(w_x, w_u) \wedge amod(w_y, w_x) \implies Triple(w_y, root(w_x), ?k)) \quad (17)$$

Consider the query *Which is the longest river in America ?.* Applying the rule in equation 16, the user triple in Listing 4 is generated.

$amod(river, longest) \wedge (longest \equiv JJS) \implies Triple(river, root(longest), ?k))$

Listing 4: User triples

```
{
River long ?x
}
```

Consider the query *Which is the most populated state in America ?*. Applying the rule in equation 18, the user triple in Listing 5 is generated.

$$\begin{aligned} & advmod(populated, most) \wedge amod(state, populated) \\ & \implies Triple(state, root(populated), ?k)) \end{aligned}$$

Listing 5: User triples

```
{
State population ?x
}
```

3.3.2. Non-scalar adjectives

These are adjectives that do not communicate the idea of scale. For a non-scalar adjective, we restrict that a syntactic constraint must match the POS tag patterns shown in figure 8. The pattern limits a non-scalar adjective to be an adjective followed directly by another adjective then a noun (e.g. female Russian astronaut) or an adjective followed directly by a noun (e.g., German chemist).

$$\boxed{JJ \rightarrow JJ \rightarrow NN | JJ \rightarrow NN}$$

Figure 8: POS tag patterns for non-scalar adjective

The POS tag pattern is translated into the rule in equation 18 and 19 respectively. The functions try to preempt all the possible modelling of the concepts' positions in the underlying ontology. Consider the statement *German fighter*. An ontology can model this in two possible ways i.e *fighter :hasNationality German* or *Germany :hasFighter fighter* hence the word *fighter* can be modelled in the subject or object position. When extracting the user triples in equation 18 and 19, we extract the two possibilities.

$$\begin{aligned}&\forall w_u, w_x, w_y.(amod(w_x, w_u) \wedge amod(w_x, w_y)\\&\implies (Triple(w_x, ?k, w_u) \wedge Triple(w_x, ?k, w_y)) \vee\\&(Triple(w_u, ?k, w_x) \wedge Triple(w_y, ?k, w_x))\end{aligned} \tag{18}$$

$$\forall w_u, w_x.(amod(w_x, w_u) \implies Triple(w_x, ?k, w_u)) \vee Triple(w_u, ?k, w_x)) \tag{19}$$

Applying the rule in equation 19 to the user query *Which German chemist won the Nobel prize ?*
$amod(Chemist, German) \implies$ *Triple(Chemist ?k German)*$\vee$ *Triple(German ?k Chemist)*
Likewise applying the rule in equation 18 to the user query *Which female German chemist won the Nobel prize*

$amod(Chemist, German) \wedge amod(female, German) \implies$

Triple(Chemist ?k German) $\wedge$ *Triple(Chemist ?k female)*$\vee$

Triple(German ?k Chemist) $\wedge$ *Triple(female ?k German)*

The user triples of a user query generated based on adjectives are added to the existing user triples generated in the previous step. For example in the query *Which female German chemist won the Nobel prize ?*, since it is a relation based query, it is first processed based on the discussion in 3.3.1, the $GenerateTriple(s)$ will extract the triples $\{$*(Chemist won Nobel)or (Nobel won Chemist)*$\}$. Adding this to the triple generated based on adjective, the final user triples for the query is shown in Listing 6.

Listing 6: User triples

```
{
(Chemist ?k German) OR (German ?k Chemist)
(Chemist won Nobel) OR (Nobel won Chemist)
}
```

3.4. Lexicon

The words in user triples need to be mapped to the entities in the underlying ontology i.e. the user triples need to speak the language of the ontology. For instance, the terms in the user triple {*State population_of ?x*} in Listing 3 need to be mapped to the terms {*State :hasPopulation ?x*} in the ontology in Listing 7. For this purpose a lexicon is developed.

Listing 7: Sample ontology

```
 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
 xml:geo="http://www.geo.net/geography/geo.owl">
 <owl:ObjectProperty rdf:ID="flow_through">
 <rdfs:domain rdf:resource="#River"/>
 <rdfs:range rdf:resource="#State"/>
 </owl:ObjectProperty>
 <owl:Class rdf:ID="River">
 <rdfs:subClassOf rdf:resource="#WaterBody"/>
 <rdfs:subClassOf>
<owl:Restriction>
<owl:onProperty rdf:resource="#hasPopulation"/>
<owl:allValuesFrom rdf:datatype="http://www.w3.org/2001/XMLSchema#
    positiveInteger"/>
</owl:Restriction>
</rdfs:subClassOf>
 </owl:Class>
 <owl:Class rdf:ID="State">
 <rdfs:subClassOf rdf:resource="#Counties"/>
 </owl:Class>
 <River rdf:ID="Mississippi_river">
 <owl:flow_through rdf:resource="#Mississippi_state"/>
 </River>
 <State rdf:ID="Mississippi_state">
 <owl:hasPopulation rdf:datatype="http://www.w3.org/2001/XMLSchema#
     positiveInteger">50000000</owl:hasPopulation>
 </State>
 <State rdf:ID="Texas">
 <owl:hasPopulation rdf:datatype="http://www.w3.org/2001/XMLSchema#
     positiveInteger">298465775</owl:hasPopulation>
 </State>
 <State rdf:ID="Georgia">
 <owl:hasPopulation rdf:datatype="http://www.w3.org/2001/XMLSchema#
     positiveInteger">4657985</owl:hasPopulation>
 </State>
 <owl:DataProperty rdf:ID="hasPopulation">
 <rdfs:domain rdf:resource="#State"/>
 <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
     positiveInteger"/>
 </owl:ObjectProperty>
</rdf:RDF>
```

The lexicon helps in mapping user terms to the entities in the ontology. A lexicon records information specific to individual entities (classes, predicates and individuals) contained in the ontology. We believe that a good lexicon should be rich enough to :

1. Disambiguate words such as Mississippi river and Mississippi state.
2. Identify positive and negative scalar adjectives in a user triple.
3. Map words in a way that minimizes the search space.
4. Resolve the exact position of a concept as modelled in the underlying

ontology.

To develop this kind of a lexicon, we adopted lemon (Lexical Model for Ontologies) (McCrae et al., 2011) which is a model for lexicons that are machine readable. It allows information to be represented relative to the underlying ontology. Lemon was a natural choice since it is RDF based and uses the principles of Linked Data. It can also be extended easily to capture the information needed. To reduce the work of generating the lexicon manually, we adopted the technique proposed in (Walter et al., 2013). We exploited the technique to generate the lexicon in lemon model semi-automatically. We designed the lexicon such that it preserved the structure of the underlying ontology. For each lexical entry in the lexicon we specify the following information:

Listing 8: Sample lexicon created from the ontology in Listing 7

```
<?xml version="1.0"?>
:state  a lemon:LexicalEntry ;
lexinfo:partOfSpeech lexinfo:noun  ;
 lexinfo:type  lexinfo:  class  ;
lexinfo:OntotripleCategory  lexinfo:  subject  ;
 lexinfo:AsscociatedPredicate  lexinfo:  hasPopulation ;
lexinfo:AsscociatedObject  lexinfo:  positiveInteger  ;
  lexinfo:positive   lexinfo:  most populated ;
  lexinfo:positive   lexinfo:  most inhabited ;
 lexinfo:negative   lexinfo:  least  populated ;
 lexinfo:negative   lexinfo:  least  inhabited  ;
  lemon:canonicalForm [ lemon:writtenRep "state"@en];
    lemon:sense [  lemon:reference
<http://dbpedia.org/ontology/state >].

 :river  a lemon:LexicalEntry ;
lexinfo:partOfSpeech lexinfo:noun  ;
 lexinfo:type  lexinfo:  class  ;
lexinfo:OntotripleCategory  lexinfo:  subject  ;
 lexinfo:AsscociatedPredicate  lexinfo:  flows_through  ;
lexinfo:AsscociatedObject  lexinfo:  state  ;
  lemon:canonicalForm [ lemon:writtenRep "river"@en];
```

```
lemon:sense [ lemon:reference
<http://dbpedia.org/ontology/river >].
```

1. *LexicalEntry* . This is a given ontology entity (class, property or individual) extracted from the underlying ontology e.g. the entity *River* in Listing 7.
2. *partOfSpeech* . This entry specifies the part of speech to which the word in the *LexicalEntry* belongs to.
3. *canonicalForm.* This is the lemma of the word in the *LexicalEntry.* To extract a canonical for of a verb, its lemma is its infinitive form or its present tense (e.g. the canonical form of the verb married is marry) . The canonical form a noun is the noun's singular form (e.g. the canonical form of the noun Rivers is River). For the adjectives it is the positive (i.e., non-negative, non-graded) form (e.g. high).
4. *type.* This is the *rdf:type* of the *LexicalEntry.*
5. *OntotripleCategory.* This indicates the position of the *LexicalEntry* in the ontology triple i.e. is it a subject, predicate or an object. If the *LexicalEntry* is a subject, then the its associated *predicate* and *object* must be specified. Likewise, if it is an object then its asscotiated *subject* and *predicate* should be indicated and finally if the lexical item is predicate then its associated *subject* and *object* should be indicated.
6. *positive* and *negative* entries. These entries are included for a *LexicalEntry* which is a domain of a *owl:DatatypeProperty* where the asscociated range is a non-negative integer. They give the positive and negative scalar adjectives associated with the *LexicalEntry* . For example in Listing 7, the entity *State* is the domain of the property *hasPopulation* and its range is a non-negative integer. Therefore its positive and negative scalar adjectives must be indicated. The positive scalar adjectives asscociated with the word *State* w are *Most Populated* and *Most Inhabited* while negative include *Least populated* and *Least Inhabited.* This entries are helpful when processing sentences that contain scalar adjectives such as *"Which is the*

most populated state in USA, the term *most populated* will be mapped to a positive scalar adjective in the lexicon.

A sample lexicon extracted from the ontology in listing 7 is shown in Listing 8.

3.5. Step 4: Converting user triples to ontology triples

The user triples need to be converted to ontology triples using the lexicon. To do this, terms in the user triples need to be mapped to the terms in the underlying ontology. To map a user triple to an ontology triple, the position a term occupies in the user triple should match the position of the term it is mapped to in the ontology triple i.e. a term in the user triple that occupies the subject position must be mapped to a term in the ontology triple that occupies subject position. By doing this, we narrow the search space hence reducing the mapping time significantly. A user triple can be in any of this forms

1. $(\langle x\rangle\langle y\rangle\langle z\rangle)$
2. $(\langle ?x\rangle\langle y\rangle\langle z\rangle)$
3. $(\langle x\rangle\langle ?y\rangle\langle z\rangle)$
4. $(\langle x\rangle\langle y\rangle\langle ?z\rangle)$

Given a user triple in the form $(\langle x\rangle\langle y\rangle\langle z\rangle)$ or $(\langle z\rangle\langle y\rangle\langle x\rangle)$ e.g. *(actor played_in Casablanca)* or *(Casablanca played_in actor)* the lexicon has to perform two key tasks.

1. When a user triple presents two options, it should select the correct triple that reflects the modelling in the underlying ontology.
2. Map user terms to ontology terms(i.e. bridge the gap between user terms and ontology terms).

Using string metric (Stoilos et al., 2005) and background knowlege Wordnet, the user entity x from the user triple $(\langle x\rangle\langle y\rangle\langle z\rangle)$ or $(\langle z\rangle\langle y\rangle\langle x\rangle)$ is mapped to the lexical entry *x_lexicon* in the lexicon. Once the term x is mapped to a term *x_lexicon* in the lexicon, the lexicon is queried to fetch all lexical entries that contain the term *x_lexicon*. We then select all *x_lexicon* where

$< ontotripleCategory >$ is the subject i.e. we select all *x_lexicon* that occupy the subject position. We then map the $< object >$ of *x_lexicon* to the $< object >$ of x in the user triple i.e. $< object >$ of *x_lexicon* is compared with $\langle z\rangle$. If they match then the terms of the user triple $\langle x\rangle\langle y\rangle\langle z\rangle)$ are mapped position by position to the lexicon terms $\langle x\_lexicon\rangle\langle y\_lexicon\rangle\langle z\_lexicon\rangle)$. The second option $(\langle z\rangle\langle y\rangle\langle x\rangle)$ is hence dropped. If they don't match the lexicon is queried to fetch *x_lexicon* that occupies the $< object >$ position. We then map the $< subject >$ of *x_lexicon* to subject in the user triple i.e. $< subject >$ of *x_lexicon* is compared with $\langle z\rangle$. If they match then the terms of the user triple $\langle z\rangle\langle y\rangle\langle x\rangle)$ are mapped position by position to the lexicon terms $\langle z\_lexicon\rangle\langle y\_lexicon\rangle\langle x\_lexicon\rangle)$. The first option $(\langle x\rangle\langle y\rangle\langle z\rangle$ is hence dropped. The same technique is applied for the various forms of user triples generated.

4. Query Construction

After converting user triples into ontology triples, the next stage is to generate a SPARQL query. The general syntax of the SPARQL query is shown in Listing 9.

Listing 9: SPARQL query structure

```
SELECT ?target1 ?target2 ...?targetn
WHERE {
Ontology triples
} Modifiers
```

The targets are the words identified in step 1 (section 3.1). For instance, consider the sentence *what is the area and population of the most populated state*, from the rule in equation (1b), the words *area* and *population* will be identified as the targets. Therefore the intial SPARQL query generated will be

```
SELECT ?area ?population
{
}
```

After identifying targets, the next step is to identify the user triples as discussed in step 2 (section 3.2). In the sentence *what is the area and population of the most populated state*, since this query is a non-relation based query, it will be processed based on the discussion in section 3.2.2. The user triple is shown in Listing 3. After generating user triples, the next step is to convert the user triples into ontology triples by the use of the lexicon as discussed in section 3.5. The triples that will be generated in our running example will be:

Listing 10: SPARQL construction

```
SELECT ?area ?population
{
State :hasPopulation ?population.
            AND
State :hasArea ?area.
}
```

For each unique noun X that appears in the subject position of the ontology triple, we add the triple $<?x> <rdf:type> <X>$ and replace the all the subsequent noun X in the ontology triples with $<?x>$. Therefore, the triple ontology triples in Listing 10 is expanded to Listing 11.

Listing 11: SPARQL construction

```
SELECT ?area ?population
{
?state rdf:type :State
?state :hasPopulation ?population.
            AND
?state :hasArea ?area.
}
```

5. Query Filters

Query filters are the additional information contained in the user submitted query that helps to further narrow down the results to meet a user's required answer. Here, we handle,

1. Logical operators (e.g. *Which river flows through Alaska or Mississippi ?*).
2. Adjectives (e.g *Which is the longest river in USA ?*).
3. Negation (e.g *Which river does not flow through Mississippi ?*).
4. Numbers (e.g. *Which are the four longest rivers in USA ?*).

5.1. Logical Operators

In SPARQL query, the conjunction(AND) operator need no special handling. Therefore, for instance in Listing 11, we just drop the AND operator resulting in the query in listing 12.

Listing 12: SPARQL construction

```
SELECT ?area ?population
{
?state rdf:type :State
?state :hasPopulation ?population.
?state :hasArea ?area.
}
```

In case of an OR operator, SPARQL provides a number of options on how to process it (SPARQL Working Group, 2013). In our case we use the UNION i.e. each identified OR operator is replaced with the key word UNION. For instance the query in Listing 13 will be transformed into 14.

Listing 13: SPARQL construction

```
SELECT ?area ?population
{
```

```
?state rdf:type :State
?state :hasPopulation ?population.
        OR
?state :hasArea ?area.
}
```

Listing 14: SPARQL construction

```
SELECT ?area ?population
{
?state rdf:type :State
{?state :hasPopulation ?population.}
    UNION
{?state :hasArea ?area.}
}
```

5.2. Adjectives

The non-scalar adjectives need no special handling apart from those discussed in section 3.3.2. However, scalar adjectives help to further narrow down the query hence need addition processing on top of those discussed in section 3.4.1. The syntactic constrains of scalar adjectives are defined in section 3.3.1. To process the scalar adjectives, we execute three steps

1. Identify a scalar adjective.
2. Categorize it as either positive or negative scalar adjective.
3. Map the adjective to a SPARQL query statement.

For the first step, the scalar adjectives are identified using the syntactic constraints defined in section 3.3.1. The scalar adjectives can either be $JSS \rightarrow NN$ or $RBS \rightarrow VBN \rightarrow NN$. If it is $JSS \rightarrow NN$, the Lexical entry of NN is queried in the lexicon to ascertain whether the JSS matches any of its $\langle$*positive*$\rangle$ or $\langle$*negative*$\rangle$ tags. For example in the query *Which is the longest river in USA ?*, the scalar adjective is *longest river*. The lexical entry of the noun *river* is queried in the lexicon to ascertain whether the term *longest* matches its $\langle positive \rangle$ or $\langle negative \rangle$ tag. If the scalar adjective is $RBS \rightarrow VBN \rightarrow NN$, the lexical entry of NN is searched to ascertain whether, $RBS \rightarrow VBN$ matches any of its $\langle positive \rangle$ or $\langle negative \rangle$ tags. For example, in the query *What is the area and population of the populated state ?*, the scalar adjective is *most populated state*. The lexical entry of the noun *state* is queried in the lexicon to ascertain whether the term *most populated* matches its $\langle positive \rangle$ or $\langle negative \rangle$ tag. If the scalar is mapped to a positive tag, the adjective is mapped to the SPAQRL statement shown in Listing 15.

Listing 15: SPARQL template for positive scalar adjective

```
SELECT DINSTINCT ?target_1 ?target_2... ?target_n
{
ontology triples
}
ORDER BY DESC(<object> of NN) LIMIT 1
```

In case of a negative tag, the scalar adjective is mapped to the SPARQL template in Listing 16.

Listing 16: SPARQL template for negative scalar adjective

```
SELECT DINSTINCT ?target_1 ?target_2...?target_n
{
ontology triples
}
ORDER BY ASC(<object> of NN) LIMIT 1
```

Therefore the final SPARQL query for the sentence *What is the area and population of the populated state ?* is shown in Listing 17.

Listing 17: SPARQL query

```
SELECT ?area ?population
{
```

```
?state rdf:type :State
?state :hasPopulation ?population.
?state :hasArea ?area.
} ORDER BY DESC(?population) LIMIT 1
```

If a user query contains both the negative and positive scalar adjectives such as in the sentence S=*Which is the longest and shortest river in America ?*, using rule in equation 11, the query will be categorized as a compound query. Therefore, the compound sentence has to be broken into two sentences, where each contains an opposing scalar adjective. For example in this query, using the rule in equation 12, we split the sentence S into
s_1=*Which is the longest river in America.*
s_2=*Which is the shortest river in America.*
cc = *and*
Each sentence is then processed independently as discussed in the previous sections to generate a SPARQL query for each sentence. For instance, s_1=*Which is the longest river in America* is processed to generate the query in Listing 18.

Listing 18: SPARQL query

```
SELECT ?river
{
?river rdf:type :River
?river :hasLength ?length
America :hasRivers ?rivers
} ORDER BY DESC(?length) LIMIT 1
```

while s_2=*Which is the shortest river in America* is processed to generate the query in Listing 19.

Listing 19: SPARQL query

```
SELECT ?river
{
?river rdf:type :River
?river :hasLength ?length
America :hasRivers ?rivers
} ORDER BY ASC(?length) LIMIT 1
```

We finally join the query as

```
SELECT *
{
{
SELECT ?river
{
?river rdf:type :River
?river :hasLength ?length
America :hasRivers ?rivers
} ORDER BY DESC(?length) LIMIT 1
}
{SELECT ?river
{
?river rdf:type :River
?river :hasLength ?length
America :hasRivers ?rivers
} ORDER BY ASC(?length) LIMIT 1


}}
```

5.3. Numbers

5.3.1. Numbered scalar adjectives

Numbered scalar adjective is a signal that a user wants a specified list of items. The syntactic constraints used to flag these type of adjectives are

$CC \rightarrow JJS \rightarrow NN$ (e.g. *four longest rivers*)

$CC \rightarrow RBS \rightarrow VBN \rightarrow NN$ (e.g. *four most populated states*)

If the scalar adjective is mapped to a positive tag of a noun (NN), the numbered adjective is mapped to the SPAQRL query shown in Listing 20

Listing 20: positive numbered adjective template

```
SELECT DINSTINCT ?target_1 ?target_2... ?target_n
{
ontology triples
}
ORDER BY DESC(<object>of NN) LIMIT (NUMBER)
```

In case it is mapped to a negative tag of a noun (NN), the scalar adjective is mapped to the query in Listing 21.

Listing 21: negative numbered adjective template

```
SELECT DINSTINCT ?target_1 ?target_2...?target_n
{
ontology triples
}
ORDER BY ASC(<object> of NN) LIMIT (NUMBER)
```

5.3.2. Numbered list

Sometimes a user query may want to get a list of items without specifying any condition such as *List four rivers in USA* or *Which four rivers flow through USA*. To identify a numbered list, we use two rules

$$\forall w_1, w_2, w_3.(num(w_1, w_2) \cap nsubj(w_3, w_1) \implies NumberedList(w_1)) \qquad (20)$$

$$\forall w_1, w_2, w_3.(num(w_1, w_2) \wedge dobj(w_3, w_1) \implies NumberedList(w_1)) \qquad (21)$$

$$(num(rivers, four) \wedge nsubj(flow, rivers) \implies NumberedList(four) \quad (22)$$

$$(num(rivers, four) \wedge dobj(List, rivers) \implies NumberedList(four) \quad (23)$$

The NumberedList is mapped to a SPARQL query in Listing 22.

Listing 22: numbered list template

```
SELECT DINSTINCT ?target_1 ?target_2...?target_n
{
ontology triples
}
LIMIT (NumberedList(w_1)) e.g. LIMIT 4
```

5.3.3. Negation

Negation is a reversal of some truth. Currently, we handle two types of negation i.e. *not* and *neither*. To recognise negation *not* we use the rule in equation 24 while to recognise *neither* we use the rule in equation 25.

$$\forall w_u, w_x, w_y.(neg(w_x, w_u \wedge nsubj(w_x, w_y) \implies Negation(w_u) \quad (24)$$

For example, *Which river does not traverse Alaska or Mississippi ?*

$$neg(not, traverse) \wedge nsubj(traverse, river) \implies Negation(not)$$

$$w_u, w_x, w_y.(advmod(w_x, w_u) \wedge nsubj(w_x, w_y) \implies Negation(w_u) \quad (25)$$

For example the query *Which river neither traverses Alaska nor Mississippi ?*

$$advmod(nor, traverses) \wedge nsubj(traverses, river) \implies Negation(nor)$$

To process the negation, we first remove the negation part and extract triples contained in the positive query. For example, in the query *Which river does* *not traverse Alaska or Mississippi ?* its positive form is *Which river traverses Alaska or Mississippi ?*. The user query is then processed normally to generate initial SPARQL query as ahown in Listing 23.

Listing 23: initial listing for negation example

```
SELECT ?river
{
{?river :run_through Mississippi}
UNION
{?river :run_through Alaska}
}
```

To handle the negation part of the user query, we extract the target words (subjects) in it using the functions in discussed in section 3.1. For each noun identified, using the lexicon, it is mapped to a corresponding term in the underlying ontology. We then extract its most general triple in the ontology. From this general triple we MINUS the triples generated by the positive form of the query. For instance the noun *river* is selected as subject of the query *Which river traverses Alaska or Mississippi ?* (see equation (1a)). After mapping the term *river* to *River* in the underlying ontology, we then extract the triple representing the general type of *River* which is of the form $<?x> <rdf:type> <X>$ i.e. $<?river> <rdf:type> <River>$. From this triple we MINUS the triple in Listing 23 to generate Listing 24.

Listing 24: listing for negation example

```
SELECT ?river
{
?river rdf:type :River
MINUS
{
{?river :run_through Mississippi}
UNION
{?river :run_through Alaska}
}
}
```

---

6. Evaluation

To gauge the performance PAROT, we evaluated it on both simple and complex questions. Simple questions are questions that can be solved using only one triple pattern (Bordes et al., 2015) while complex questions are questions which the intended SPARQL query consist more than one triple pattern (Trivedi et al., 2017). By using these two categories of datasets, we sought to evaluate if the extra capabilities of PAROT gives it any performance advantage over tools that don't incorporate techniques such as negation, adjective and compound sentence handling capabilities.

6.1. Dataset

For complex questions, we used two datasets, the first dataset was from the 9th challenge on question answering over linked data (QALD-9)[2]. We specifically evaluated the PAROT system on the test dataset. The questions contained in the datasets are of different complexity, including questions with counts, superlatives comparatives and temporal aggregators. The second type of dataset used was that provided by Mooney[3] which has been used previously by PANTO for similar evaluation. We specifically used the dataset that is composed of geography data in the United States. The dataset is accompanied with 880 queries where each query has its expected response in prolog format. From this dataset, we converted the prolog format into OWL ontology. We then selected queries that were compound in nature and contained negation. To evaluate the performance of PAROT on simple questions, we used 200 questions and their corresponding answers as proposed by (Bordes et al., 2015).

[2]https://github.com/ag-sc/QALD/tree/master/9/data.

[3]ftp://ftp.cs.utexas.edu/pub/mooney/nl-ilp-data/geosystem/geobase

6.2. Evalutation Metrics

To evaluate the PAROT system, we replicated the metrics used in QALD-9 challenge. Specifically, we used the following parameters

$$precision = \frac{\text{Number correct answers generated for question q}}{\text{Number of total answers generated for question q}} \tag{26}$$

$$recall = \frac{\text{Number correct answers generated for question q}}{\text{Number of gold standard answers provided for question q}} \tag{27}$$

$$F - measure = \frac{2\times\text{recall}\times\text{ precision}}{\text{recall}+\text{ precision}} \tag{28}$$

We also adopted the following rules

1. If for a given question q the gold answerset is empty and the system also generates an empty answerset, the precision, recall and F-measure values is set to 1.
2. If for a given question q the gold answerset is empty but the system generates an answer, the precision, recall and F-measure values is set to 0.
3. If for a given question q the system generates an empty answer set while the gold answerset is not empty the precision is set to 1 while the recall and F-measure values are set to 0.

We then computed the macro and micro F-measure of PAROT over all test questions. To compute micro-F-measure, we summed up all true and false positives and negatives and calculated the precision, recall and F-measure at the end. For the macro-measures, we calculated precision, recall and F-measure per question and averaged the values at the end. The results were compared with those of gAnswer tool (Zhao et al., 2017), which was the top performing tool in QALD-9 challenge (Usbeck et al., 2018a).

Table 2: PAROT vs gAnswer in Macro results

Tool	Macro-Precision	Macro-Recall	Macro-F-measure	Query average processing time(ms)
PAROT	0.8107	0.7432	0.7755	163
gAnswer	0.83	0.729	0.7763	154

Table 3: PAROT vs gAnswer in Micro results

Tool	Micro-Precision	Micro-Recall	Micro-F-measure	Query average processing time(ms)
PAROT	0.8256	0.7901	0.8075	163
gAnswer	0.84	0.7621	0.79915	154

6.3. Results and Discussion

6.3.1. Simple questions results

From the results in table 2 and 3 gAnswer performs slightly better in terms of precision in both micro and macro-precision. However, when it comes to recall PAROT performs slightly better in both Micro and micro-recall. From the F-measure values the tools are almost equal. From these results when dealing with simple questions, the development of PAROT may not be justified even though it is able to fetch more correct answers as compared to gAnswer as depicted by its slightly higher recall value.

6.4. Complex questions results

6.4.1. Results for QALD9 dataset

When dealing with complex questions, the strength of PAROT is evident. In QALD9 dataset, the PAROT outperforms gAnswer in both macro F-measure

Table 4: PAROT vs gAnswer in Macro results

Tool	Macro-Precision	Macro-Recall	Macro-F-measure	Query average processing time(ms)
PAROT	0.4321	0.512	0.4687	3577
gAnswer	0.293	0.327	0.3091	2908

Table 5: PAROT vs gAnswer in Micro results

Tool	Micro-Precision	Micro-Recall	Micro-F-measure	average total processing time(ms)
PAROT	0.5610	0.5765	0.5686	3577
gAnswer	0.4009	0.4112	0.4060	2908

and micro F-measure as shown in table 4 and 5. This is attributed to its ability to handle a number of variety of questions i.e. the wide coverage of the syntactic based heuristics. PAROT performs 18% better that gAnswer in this task. Its high coverage is depicted by its comparatively higher recall value. Its high precision shows that the heuristics are able to resolve user questions into correct SPARQL queries. However, an optimum performance of PAROT was inhibited by its inability to answer questions that start with *When*. It also can only partially handle aggregation. When it comes to query processing time qAnswer has a significant lower response time as compared to PAROT. The significantly slow query response time of PAROT is attributed to its elaborate query analysis and categorization step which takes a significant amount of time as compared to query conversion to SPARQL and answer retrieval steps. The query processing time does not include the time for loading ontology and parsing it to create the lexicon.

6.4.2. Results of Geoquery dataset

Table 6: PAROT vs gAnswer in Macro results

Tool	Macro-Precision	Macro-Recall	Macro-F-measure	Query average processing time(ms)
PAROT	0.8512	0.8971	0.8755	1988
gAnswer	0.6711	0.6842	0.6776	1606

Table 7: PAROT vs gAnswer in Micro results

Tool	iacro-Precision	Micro-Recall	Micro-F-measure	Query average processing time(ms)
PAROT	0.8763	0.8901	0.8831	1988
gAnswer	0.6524	0.7013	0.6760	1606

In Geoquery dataset which we tailored to contain mostly negation and compound queries, PAROT still outperforms gAnswer both in macro and micro F-measure as shown in table 6 and 7. It achives a macro F-measure of 87.55% as compared to gAnswer's 67.76%. When it comes to micro F-measure, PAROT achieves an F-measure of 88.31% as compared to qAnswer's 67.60%. This is an indication that PAROT is significantly efficient when handling compound and negation based questions. Some of the wrong answers generated by PAROT were attributed to wrong dependency relationships generated by Stanford dependency parser.

6.4.3. Weaknesses of PAROT

From the evaluation of PAROT, we noted that it had the following weaknesses which we seek to address as an ongoing work;

1. Its query analysis and categorization step is significantly slow.

2. It still cannot handle questions that start with *When*.
3. It still has a low precision and recall when processing aggregation based questions.
4. The lexicon generation step is still slow hence not scalable to large ontologies.

7. Conclusion

PAROT is a NL to SPARQL tool. It has the ability to handle compound, negation, numbered list and scalar adjective based questions in addition to other questions. PAROT adopts an approach that generates the most likely triple from the user query which is then validated by the lexicon. It relies on dependency parser to process user's queries to user triples. The user triples are then converted to ontology triples by the lexicon. The triples generated by the lexicon is what is used to construct SPARQL query that fetches the answers from the underlying ontology. Based on the results discussed in section 6, PAROT is highly successful in converting NL to SPARQL. It can therefore be used in a system that anticipates to bridge the gap between users and ontologies. As an ongoing work we seek to use PAROT to develop an ontology based chatbot for diagnosing chicken diseases.
Source code for PAROT implementation is released at[4].

References

A. Fader, S. Soderland, O. E. (2011). Identifying relations for Open Information Extraction. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 1535–1545.). Edinburgh, UK. doi:10.1016/j.jcrysgro.2013.04.025.

Ahmad, S., & Hunt, B. J. (2015). Effective Approaches to Attention-based Neural Machine Translation. In *In Proceedings of the 2015 Conference on*

[4]https://github.com/onexpeters/PAROT_MASTER/tree/master/src

Empirical Meth- ods in Natural Language Processingn September (pp. 1412–1421). doi:10.1007/978-3-319-28308-1_29.

Bordes, A., Usunier, N., Chopra, S., & Weston, J. (2015). Large-scale Simple Question Answering with Memory Networks. *CoRR*, . URL: http://arxiv.org/abs/1506.02075. arXiv:1506.02075.

Bouziane, A., Bouchiha, D., Doumi, N., & Malki, M. (2015). Question Answering Systems: Survey and Trends. *Procedia Computer Science*, *73*, 366–375. URL: http://dx.doi.org/10.1016/j.procs.2015.12.005. doi:10.1016/j.procs.2015.12.005.

Cai, R., Xu, B., Zhang, Z., Yang, X., Li, Z., & Liang, Z. (2018). An encoder-decoder framework translating natural language to database queries. *IJCAI International Joint Conference on Artificial Intelligence*, *2018-July*, 3977–3983.

Cimiano, P., & Bielefeld, U. (2011). Is Question Answering fit for the Semantic Web ?: a Survey . *Semantic Web*, *2*, 125–155.

Cunningham, H., Maynard, D., Bontcheva, K., & Tablan, V. (2001). Gate. *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*, (p. 168). URL: http://portal.acm.org/citation.cfm?doid=1073083.1073112. doi:10.3115/1073083.1073112.

Damljanovic, D., Agatonovic, M., & Cunningham, H. (2011). FREyA : An Interactive Way of Querying Linked. *Eswc*, (pp. 125–138).

Diefenbach, D., Both, A., Singh, K., & Maret, P. (2018). Towards a Question Answering System over the Semantic Web. *Semantic Web*, *0*, 1–15. URL: http://arxiv.org/abs/1803.00832. arXiv:1803.00832.

Ferré, S. (2017). Sparklis: An expressive query builder for SPARQL endpoints with guidance in natural language. *Semantic Web*, *8*, 405–418. doi:10.3233/SW-150208.

Gur, I., Yavuz, S., Su, Y., & Yan, X. (2018). DialSQL: Dialogue Based Structured Query Generation. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, (pp. 1339–1349). URL: `https://www.aclweb.org/anthology/papers/P/P18/P18-1124/`.

Hao, Y., Zhang, Y., Liu, K., He, S., Liu, Z., Wu, H., & Zhao, J. (2017). An End-to-End Model for Question Answering over Knowledge Base with Cross-Attention Combining Global Knowledge. In *In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics* (pp. 221–231). doi:`10.18653/v1/p17-1021`.

He, S., Zhang, Y., Liu, K., & Zhao, J. (2014). CASIA@V2: A MLN-based Question Answering System over Linked Data. In *CLEF* 61272332 (pp. 1249–1259).

Hochreiter, Sepp and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*, 1–32.

Iyer, S., Konstas, I., Cheung, A., Krishnamurthy, J., & Zettlemoyer, L. (2017). Learning a Neural Semantic Parser from User Feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. URL: `http://arxiv.org/abs/1704.08760`. `arXiv:1704.08760`.

Kaufmann, E., Bernstein, A., & Zumstein, R. (2006). Querix: A Natural Language Interface to Query Ontologies Based on Clarification Dialogs. In *In proceedings of the 5th International Semantic Web Conference (ISWC 2006)* November (pp. 5–6). Athens, GA.

Klein, D., & Manning, C. D. (2003). Accurate unlexicalized parsing. *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - ACL '03*, *1*, 423–430. URL: `http://portal.acm.org/citation.cfm?doid=1075096.1075150`. doi:`10.3115/1075096.1075150`.

Lehmann, J., & Bühmann, L. (2011). AutoSPARQL: Let users query your knowledge base. *Lecture Notes in Computer Science (including subseries Lec-*

ture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), *6643 LNCS*, 63–79. doi:10.1007/978-3-642-21034-1_5.

Lopez, V., Pasin, M., Motta, E., Hall, W., & Keynes, M. (2005). AquaLog : An Ontology-Portable Question Answering System for the Semantic Web. *In Proceedings of the 2008 Conference on Semantics in Text Processing*, (pp. 546–562). doi:10.1007/11431053_37.

Marneffe, M.-c. D., & Manning, C. D. (2015). Stanford typed dependencies manual. *20090110 Httpnlp Stanford*, *40*, 1–22. URL: http://nlp.stanford.edu/downloads/dependencies{_}manual.pdf. doi:10.1.1.180.3691.

McCrae, J., Spohr, D., & Cimiano, P. (2011). Linking lexical resources and ontologies on the semantic web with lemon. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (pp. 245–259). volume 6643 LNCS. doi:10.1007/978-3-642-21034-1_17.

Sander, M., Waltinger, U., Roshchin, M., & Runkler, T. (2014). Ontology-Based Translation of Natural Language Queries to SPARQL. *2014 AAAI Fall Symposium Ontology-Based*, (pp. 42–48). doi:10.1016/j.ijmedinf.2012.01.005.

SPARQL Working Group (2013). SPARQL Query Language for RDF. URL: http://www.w3.org/TR/rdf-sparql-query/.

Stoilos, G., Stamou, G., & Kollias, S. (2005). A string metric for ontology alignment. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *3729 LNCS*, 624–637. doi:10.1007/11574620_45.

Tatu, M., Balakrishna, M., Werner, S., Erekhinskaya, T., & Moldovan, D. (2016). Automatic Extraction of Actionable Knowledge. *Proceedings - 2016 IEEE 10th International Conference on Semantic Computing, ICSC 2016*, (pp. 396–399). doi:10.1109/ICSC.2016.29.

Trivedi, P., Maheshwari, G., Dubey, M., & Lehmann, J. (2017). LC-QuAD: A corpus for complex question answering over knowledge graphs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *10588 LNCS*, 210–218. doi:10.1007/978-3-319-68204-4_22.

Usbeck, R., Gusmita, R. H., Saleem, M., & Ngomo, A. C. N. (2018a). 9th challenge on question answering over linked data (QALD-9). *CEUR Workshop Proceedings*, *2241*, 58–64. doi:10.1007/978-3-319-69146-6_6.

Usbeck, R., Gusmita, R. H., Saleem, M., & Ngomo, A. C. N. (2018b). 9th challenge on question answering over linked data (QALD-9). *CEUR Workshop Proceedings*, *2241*, 58–64. doi:10.1007/978-3-319-69146-6_6.

Walter, S., Unger, C., & Cimiano, P. (2013). A corpus-based approach for the induction of ontology lexica. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *7934 LNCS*, 102–113. doi:10.1007/978-3-642-38824-8_9.

Wang, C., Xiong, M., Zhou, Q., & Yu, Y. (2007). PANTO: A Portable Natural Language Interface to Ontologies. *The Semantic Web: Research and Applications*, (pp. 473–487). URL: http://link.springer.com/10.1007/978-3-540-72667-8{_}34. doi:10.1007/978-3-540-72667-8_34.

Xu, X., Liu, C., & Song, D. (2017). SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning. . *CoRR*, (pp. 1–13). URL: http://arxiv.org/abs/1711.04436. arXiv:1711.04436.

Yahya, M., Berberich, K., & Elbassuoni, S. (2012). Natural Language Questions for the Web of Data. In *EMNLP* July (pp. 379–390). Deanna12.

Yu, T., Yasunaga, M., Yang, K., Zhang, R., Wang, D., Li, Z., & Radev, D. (2018). SyntaxSQLNet: Syntax Tree Networks for Complex and Cross-DomainText-to-SQL Task. In *In Proceedings of the 2018 Conference on*

Empirical Methods in Natural Language Processing (pp. 1653–1663). URL: `http://arxiv.org/abs/1810.05237`. `arXiv:1810.05237`.

Zhao, D., Zou, L., Wang, H., Yu, J. X., & Hu, S. (2017). Answering Natural Language Questions by Subgraph Matching over Knowledge Graphs. *IEEE Transactions on Knowledge and Data Engineering*, *30*, 824–837. doi:`10.1109/tkde.2017.2766634`.