ASQFor: Automatic SPARQL Query Formulation for the Non-Expert

Muhammad Rizwan Saeed a,∗, Charalampos Chelmis b and Viktor K. Prasanna a

a Ming Hseih Department of Electrical Engineering, University of Southern California, CA, USA
E-mail: saeedm@usc.edu
b Department of Computer Science, University at Albany - SUNY, NY, USA
E-mails: cchelmis@albany.edu, prasanna@usc.edu

Abstract. The combination of data, semantics, and the Web has led to an ever growing and increasingly complex body of semantic data. Accessing such structured data requires learning formal query languages, such as SPARQL, which poses significant difficulties for non-expert users. To date, many interfaces for querying Ontologies have been developed. However, such interfaces rely on predefined templates or require expensive pre-processing and customization. Natural Language (NL) interfaces are particularly preferable to other interfaces for providing users with access to data. However the inherent difficulty in mapping NL queries to semantic data can create ambiguities during query formulation phase. To avoid the pitfalls of existing approaches, while at the same time retaining the ability to capture users’ complex information needs, we propose a simple keyword-based search interface to the Semantic Web. Specifically, we propose Automatic SPARQL Query Formulation (ASQFor), a systematic framework to issue semantic queries over RDF repositories using simple concept-based search primitives. ASQFor has a very simple interface, requires no user training, and can be easily embedded in any system or used with any semantic repository without prior customization. We demonstrate via extensive experimentation that ASQFor significantly speeds up query formulation while at the same time matching the syntax of hand-crafted queries.

Keywords: Semantic Web, Automatic Query Formulation, Ontologies, SPARQL, RDF

1. Introduction

The suite of technologies developed for realizing the Semantic Web such as Ontologies, Semantic Annotations and Linked Data can be used for modeling, integration, querying, and sharing of information on the Web [1]. In recent years, the Semantic Web standards have evolved. The improvements and innovations in this field have allowed the delivery of more complex, more sophisticated and more far-reaching semantic applications in information brokering, knowledge management, and decision support in diverse fields such as telecommunications, logistics, manufacturing, energy, health, tourism, publishing, and culture [38]. As more and more semantic data become available, the question of how end users can access this body of knowledge becomes of crucial importance. Tools for creating, editing, and querying Ontologies have been widely developed however accessing semantic data requires intimate familiarity with existing formal query languages such as SPARQL1. Despite their strong expressive power, such formal languages impose an initial barrier to adoption due to their hard requirement for understanding of their formal syntax and how knowledge is encoded in semantic repositories.

The Resource Description Framework2 (RDF) Semantic Web standard and its semantic query language, SPARQL, have been recognized as one of the key technologies of the Semantic Web. An RDF repository is a collection of triples (denoted as <subject, predicate, object>) which can be represented as a graph, the vertices of which denote subjects and objects and edges denote predicates. SPARQL allows users to write queries against data repositories that follow the RDF specification of the World Wide Web Consortium (W3C) by creating queries that consist of triples, conjunctions, disjunctions, and optional patterns. Although SPARQL is a standard way to access RDF data, it remains tedious and difficult for end-users

1http://www.w3.org/TR/rdf-sparql-query/
2http://www.w3.org/RDF/
N. Surname1 et al. / ASQFor: Automatic SPARQL Query Formulation for the Non-Expert

(a) Schema Ontology for University Data

(b) Sample SPARQL Query

Fig. 1. Running Example of SPARQL Query Formulation

because of the complexity of its syntax and the RDF schema [29].

1.1. Challenges

Consider our running example shown in Figure 1. The example query corresponds to the natural language question: “What are names of the students taking the course CS570 and name of the professor teaching it?”. Figure 1b illustrates the hand-crafted semantic query that returns the correct result. To automatically generate such a SPARQL query, a system would have to (i) separate the input into syntactic markers and “meaningful” tokens, (ii) map tokens to concepts in the Ontology, (iii) link identified concepts based on relationships in the Ontology, and (iv) issue the query to collect the results.

An ideal system would allow end-users to benefit from the expressive power of Semantic Web standards while at the same time hiding their complexity behind an intuitive and easy-to-use interface [16, 21]. Therefore, significant attention to interfaces for querying semantic repositories has resulted in a wide range of systems across disciplines including Natural Language Processing (NLP) systems [8, 19, 36, 37], Semantic Web Technologies [16, 23, 40], and visualization environments [18, 29, 32].

Modern query languages for the Semantic Web do not really support the handling of natural language text, requiring specialized solutions ranging from predefined templates which provide the skeleton for SPARQL queries [29, 35] to quasi natural language querying systems [3, 11, 16] which rely on controlled vocabularies to guide the user step-by-step through the set of possible queries with suggestions of terms that are connected in the Ontology. While such approaches make Ontology queries more straightforward, (i) they require expensive customization to each new domain or Ontology and (ii) adding new templates requires the involvement of domain experts and language engineers. Furthermore, Natural Language interfaces can be limited by ambiguity and even with controlled vocabularies they require adherence to specific syntactic or grammatical rules.

Conversely, keyword-based search over hypertext documents is an established technology that is being used by search engines to capture users’ complex information needs. In fact, search engines have become popular because of their simplistic conceptual model, i.e., results include those documents that match the specified keywords. Such concept-based queries can be used to capture the information needs of a query (e.g. “Graduate Students CS570”) while at the same time offering a Google-like search box interface to the end-user.

To summarize, a system that aims to abstract the complexities of Semantic Web standards to allow user to interact with semantic data needs to achieve the following targets:

1. Minimum reliance on predefined rules: Systems that rely on static dictionaries or pre-defined rules may lack portability and require customization before they can adjust to changes in schema Ontology of the semantic repository or can be used with a different one altogether.

2. Minimum query formulation overhead: The query formulation overhead can come from processing user inputs or creating and updating dictionaries, rules or templates needed for query formulation.

3. Easy to use interface: Since the purpose of the querying abstraction (or interface) is to provide an alternative to formulating SPARQL queries di-
rectly, hence it should be simpler and easier to use than the actual SPARQL standard but at same time provide as much functionality as possible. 

(4) **Scalable Approach**: The query formulation approach should be able to scale with the size of the schema Ontology or number of triples in the semantic repository.

### 1.2. Our Approach

In this paper, we take a `<key,value>` approach to the problem of querying a semantic data repository. For example, the equivalent keyword-based query for Figure 1 is `<Name,*>, <Professor, *>, <GradStudent, *>, <courseName, CS570>`, which is similar to the way arguments are passed to functions in programming languages such as Java. With Automatic SPARQL Query Formulation (ASQFor) algorithm, our aim is to create a reusable, extendable, and domain independent approach that can be used by users to query RDF repositories with virtually no training and prior understanding of Semantic Web. ASQFor’s simple and intuitive tuple-based interface accepts `<key,value>` inputs and translates them into a formal language query (currently SPARQL). Generated queries are then executed against the semantic repository and the result is returned to the user. The users, thus, only needs to be aware of the available information hosted by the triple store and formulate their search criteria in terms of key-value pairs consisting of relevant terms and filtering values.

Our main contributions are:

1. Develop a domain independent framework that provides a simple but powerful way of specifying complex queries and automatically translates them into formal queries on the fly (i.e., does not rely on predefined rules and can instantaneously adapt to changes in the Ontology).

2. Using real-world data, we evaluate ASQFor both (i) quantitatively to indicate possible performance overheads, and (ii) qualitatively to identify the possible ease-of-use and increased productivity in information searching activities as a direct result of reducing the amount of time spend to manually develop and adapt queries.

### 2. Related Work

In the domain of Semantic Web, Ontologies (or vocabularies) define the concepts and relationships used to describe and represent an area of concern [2]. The purpose of creating the Ontologies and integrating data is to organize heterogeneous data sources for simplifying on-demand information access and enabling complex analytics to be performed on the integrated knowledge base. In recent years, there has been an increase in the availability of semantic data. Search engines such as Google [31] embed Linked Data in Web pages using Extensible Hypertext Markup Language (XHTML) with RDFa mark-ups. The American Art Collaborative (AAC), a consortium of 14 museums in the United States, allows public access to Linked Open Data on the subject of American Art through SPARQL endpoint which serves as a research tool for scholars and curators and as a public interface for students, teachers, and museum visitors [33]. Another example is from the domain of sensor networks, where data from various sensors and associated metadata are being integrated using the principles of Semantic Web. In this context, Semantic Web technologies have been used in the areas of oceanographic measurements [5], ecological surveys [22], smart grids [42], and external corrosion monitoring [28]. These applications have benefited from the Semantic Web based approaches in terms of discovery, contextualization, and integration of data. Such representation not only explicitly reveals relationships between facts but can also be used to drive methods to infer hidden or implicit relationships between entities. For instance, it is shown by authors in [30] that certain queries are only possible due to reasoning capabilities over semantic data and cannot be issued over relational data using SQL. Once the semantic data has been made available, the end user must have the pre-requisite skills to be able to effectively utilize the data i.e., the knowledge of the exact structure of the schema Ontology and SPARQL syntax.

Developing Semantic Web applications require handling of RDF concepts and data in a programming language [27]. Currently majority of software is developed using object-oriented programming languages, however programming in RDF is triples-oriented. Attempts to integrate Semantic Web and object oriented programming have thus far resulted in solutions in which there is always a trade-off between cost, performance, and simplicity of use [24, 25]. Such solutions

---

3http://www.w3.org/TR/rdfa-syntax/
Table 1
Literature Survey

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Approach</th>
<th>No pre-processing</th>
<th>No user guided dis-ambiguation</th>
<th>Handles subclass relationships</th>
<th>No restrictions on structure of ontology</th>
<th>No limitations on query size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword-based Approach</td>
<td>ASQFor</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Constrained NL Input</td>
<td>Ginseng [3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Querix [17]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>SQUALL [11]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>NL Input</td>
<td>Aqualog [20]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Auto SPARQL [19]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NLP-Reduce [15]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PANTO [37]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pradel et al. [26]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sander et al. [29]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sparklis [12]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Unger et al. [35]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>USI Answers [36]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Zheng et al. [41]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

that propose modification of programming syntax require introduction of new compilers and interpreters.

In order to understand users’ information needs accurately enough to allow for retrieving a precise answer, interfaces that translate users’ Natural Language (NL) based queries into formal queries have been explored [8, 11, 19, 29, 36, 37]. Compared to keyword-based search, systems based on natural language can imply semantic relationships between keywords using a whole sentence [13]. However, simple search-box and concept-based search interfaces have been shown to achieve comparable results to NL query approaches [9]. Additionally, some existing natural language based approaches limit input to a subset of natural language rules by introducing a pre-specified vocabulary, grammar, or sentence structures that must be followed while constructing a query [12].

Approaches that avoid the challenges of natural language processing rely on controlled environments by guiding the user step by step with suggestions of terms that are connected in the Ontology [3, 7, 17], formulating queries interactively. Querix [17] translates natural language queries into SPARQL. In case if the NL query translates into multiple semantic queries then Querix relies on clarification from the users via dialog boxes. Ginseng (Guided Input Natural language Search Engine) [3] allows users to query OWL knowledge bases using a controlled input language. The system provides suggestions through pop-up lists for each word in the user entry. These pop-up menus offer suggestions on how to complete the current user entered word and show the options for the next word. The possible choices get reduced as the user continues to type. Entries that are not part of these suggested lists are not accepted by the system. Once a query is generated, Ginseng translates the entry to SPARQL statements, executes it against the Ontology model and displays the SPARQL query and answer to the user.

Approach by Sander et al. [29] requires predefined SPIN (SPARQL Inferencing Notation) rules to be stored in the semantic repository. SPIN rules are essentially SPARQL statements stored as part of the RDF graph. Similarly, form-based query construction methods [12], require users to fill out a variety of information in web forms, which may be both cumbersome and time consuming. Approaches that rely on predefined rules limit the ability of users to formulate new queries on demand and rely on involvement of IT experts or database admins to add new queries to the library [18].

Finally, approach presented in [34] adds a step related to graph summarization since it works on the data graph to construct queries. Another step is added for pre-processing user inputs using lexical databases (i.e., Wordnet\footnote{https://wordnet.princeton.edu/}). This results in construction of multiple queries and hence the approach presents the “top-k queries” to the user to select one for retrieving all answers. Instead, our proposed approach reduces the computational requirements of search while at the
same time enabling queries that contain semantic relationships which are represented by a path in the RDF graph.

Table 1 summarizes the comparison of different approaches that aim to translate users’ query intentions into formal SPARQL queries. Usually the approaches with NL or constrained NL inputs require pre-processing to disambiguate the user input. During the query formulation stage, some approaches rely on pre-computed dictionaries and rules such as templates, SPIN rules, and grammar rules [10, 11]. Some approaches [20] limit input to NL queries that map to few triples only. ASQFor doesn’t require pre-processing and does not rely on pre-computed rules for formulating queries. The schema Ontology is traversed on the fly to formulate SPARQL queries, resulting in a minimum formulation overhead. Since a user query does not need to be linguistically correct but must contain a minimum set of “relevant” concepts, we believe that our proposed keyword-based is more suitable for querying triple stores. This eliminates the need of pre-processing natural language phrases into discernible tokens before matching such keywords to concepts and attributes in the Ontology. If a user requires results based on multiple attributes from the database, then it will be tedious to formulate such a query through an NL interface. ASQFor can also handle subclass relationships among different classes. Hence ASQFor provides an alternative to the approaches discussed above with a different set of strengths and limitations, which we discuss in detail in next section.

3. ASQFor: Automatic SPARQL Query Formulation

The main goal of ASQFor is to enable end-users to formulate semantic queries in terms of classes and data properties while being oblivious to the actual structure of the semantic data.

3.1. Algorithm

Let $G$ be the graph of schema Ontology with the root $r$ and let $Q$ be the query subgraph with the root $\hat{r}$. The work flow of the algorithm is as follows:

1. The user provided list of keywords (such as <Name, *>, <Professor, *>, <GradStudent, *>, <courseName, CS570>) is modified so that all the attributes (data properties) are replaced by their respective domain classes. This is computed by the function $domain(k)$, which is invoked in Lines 5, 14, and 33 in Algorithm 1. The function takes as input the URI of a data property $k$, and outputs the URI of its domain class by issuing the SPARQL query shown in Figure 2. The SPARQL statements for these data properties are generated in the final step. For each identified domain class, the algorithm keeps track of all of its direct and indirect subclasses which can inherit its data attributes. From our running example (Figure 1), the attribute name is associated with class Person. All the direct and indirect subclasses of Person inherit this attribute e.g. Student, PhDStudent, Professor etc. If name is part of the user provided list of keywords, then all of those (direct or indirect) subclasses of Person that also appear in the input list will have separate SPARQL statements in the final query linking them with the attribute name (see statements 3 and 4 in Figure 3).

2. ASQFor makes rooted tree assumption which means that the root $r$ of the schema Ontology $G$ is unique and can be determined using a single SPARQL query. Path from each class in the modified input list to the root $r$ is traced in order to compute LCA (Lowest Common Ancestor) of all the domain classes in the modified input list. The computed LCA is the root node $\hat{r}$ of the final query subgraph $Q$.

3. The algorithm iterates through the list of classes (from step 1). During each iteration, a node is selected from the list and is marked visited. In order to generate correct SPARQL statement, the algorithm determines if the currently selected node is range of a user-defined object property, a subclass of another class or both. In the first and third cases, the algorithm traces the path towards $\hat{r}$ using the domain of the identified user-defined object property and generate corresponding SPARQL statement using the current node,

SELECT ?domain WHERE {
  FILTER NOT EXISTS (<http://dataprop1> rdfs:range ?any)
}

Fig. 2. SPARQL query to get domain of a data property
the object property of which it is range and domain of that object property. In the second case
where the current node is subclass of another class, the query variable assigned to the current
node is assigned to its superclass. This process is repeated until the root \( r \) of the query subgraph \( Q \)
or an already visited node is reached, after which the next keyword is selected from the modified
input list and the process is repeated.

(4) The final step is to link the data properties to their respective domain classes through SPARQL
statements and create filtering statements.

3.2. Lowest Common Ancestor

To determine LCA (which will be root \( r \) of \( Q \)), we traverse \( G \), starting from the classes identified in the
user input, towards the root \( r \). The information about the predecessors of selected nodes at each steps is ac-
bquired by using SPARQL queries. We start by determin-
ing the common ancestor of two given nodes \( u \) and \( v \). The path from \( u \) to \( r \) and from \( v \) to \( r \) are computed.
Both paths are compared to find the LCA for these two
nodes. We, then, select the next node from the user in-
put and find LCA of the selected node and the LCA
computed in previous step. Algorithm 2 determines the
root \( r \) of the query subgraph by repeatedly making use of
Algorithm 3 to compute pairwise LCA of any two
given nodes.

3.3. Comprehensive Example

Consider the keyword-based query:

\(<\text{Name},*,*>,<\text{GradStudent},*,*>,<\text{Professor},*>\),<\text{courseName},"CS570" >

The targets of the query are different concepts and attributes that lie on different branches of \( G \)
e.g. the attributes name and courseName and the classes GradStudent and Professor. To formulate
the SPARQL query, it is important to know how these concepts and attributes are related to each other. In
our example name is an attribute of Person which is

\[1. \text{?gradstudent university:takesCourse ?course.}\]
\[2. \text{?course university:isTaughtBy ?professor.}\]
\[3. \text{?professor university:name ?professorname.}\]
\[4. \text{?gradstudent university:year ?gradyear.}\]
\[5. \text{?course university:courseName ?coursename.}\]

Fig. 3. Step-wise generated SPARQL statements

Algorithm 1 ASQFor(\( L \))

Input: list \( L \) of key value pairs \( <K,V> \)

Output: SPARQL query \( Q \) that encapsulates the keywords provided by the user and their semantic relationships
that are inferred by the Ontology. In case values are provided, filtering statements are also included to ensure the
information need of the end-user is met.

1: \( Q, varDictionary \leftarrow \emptyset \)
2: for each key-value pair \( <k,v> \in L \) do
3: add variable for \( k \) in \( varDictionary \)
4: if \( k \) is a data property then
5: add variable for \( \text{domain}(k) \) in \( varDictionary \)
6: end if
7: if \( v = \emptyset \) then
8: insert \( k \) in the query header
9: end if
10: end for
11: \( r \leftarrow \text{findLCA}(L) \)
12: for each key-value pair \( <k,v> \in L \) do
13: if \( k \) is a data property then
14: currentNode \leftarrow \text{domain}(k)
15: else
16: currentNode \leftarrow k
17: end if
18: while (currentNode.visited == 0 and currentNode \( \neq r \)) do
19: currentNode.visited \leftarrow 1
20: if \( \exists \) triple \( <\text{class},prop,\text{currentNode}> \) in \( G \) then
21: classVar \leftarrow varDictionary.get(class)
22: cNodeVar \leftarrow varDictionary.get(currentNode)
23: \( Q \leftarrow \text{insert triple } <\text{classVar},prop,cNodeVar> \)
24: else
25: if \( \exists \) triple \( <\text{currentNode},\text{rfds}: \text{subClassOf},\text{class}> \) in \( G \) then
26: childVar \leftarrow varDictionary.get(currentNode)
27: insert (or replace) pair \( (\text{class},\text{childVar}) \) in \( varDictionary \)
28: end if
29: end if
30: currentNode \leftarrow class
31: end while
32: if \( k \) is a data property then
33: \( Q \leftarrow \text{insert triple } <\text{domain}(k),prop,k> \)
34: if \( v \neq \emptyset \) then
35: \( Q \leftarrow \text{insert filter statement for } k \text{ using } v \)
36: end if
37: end if
38: end for
39: return \( Q \)
the superclass of both GradStudent and Professor. 
courseName is an attribute of class Course, which is 
related to the concepts Professor and GradStudent 
through properties isTaughtBy and takesCourse 
respectively.

In the first step of the algorithm, all the data properties 
(name and courseName) are resolved to their respective 
domains (Person and Course respectively). The modified 
input list of keywords then contains: Person, GradStudent, 
Professor, and Course, where each entry is of type Class. 
In order to establish the relationship between these nodes 
the algorithm finds the smallest subgraph \( Q \) that connects 
them. To do so we find their lowest common ancestor, 
which becomes the root \( r \) of the query subgraph \( Q \). 
In this example, \( r \) and \( f \) are the same i.e., Person.

In the first iteration of step 3 of the algorithm, 
Person is selected. However no further processing 
is done as it is the root \( r \) of the query subgraph \( Q \). 
In the next iteration GradStudent is selected. 
The algorithm determines that it is subclass of 
another class Student. So it assigns the variable for the 
subclass GradStudent, say \(?gradstudent\) to class 
Student. Any SPARQL statement generated relevant 
to this class in current iteration must use the variable 
\(?gradstudent\). Using Student, the algorithm selects 
Course as the next node in the path to the root 
\( r \). Using Student, Course and their linking property 
takesCourse, the algorithm generates the SPARQL 
statement 1 shown in Figure 3. Note that the variable 
\(?gradstudent\) is used for Student.

With Course as currently selected node, the next 
node on path to root \( r \) is Professor through the object 
property isTaughtBy. Hence statement 2 (from Figure 3) 
is generated. With Professor as the next selected 
node, the algorithm determines that it is the 
subclass of Person. The update of variable dictionary 
occurs as before, however, no new statement will be 
generated for Person in this iteration as it is the root 
\( r \) of the query subgraph \( Q \). For the next class nodes 
Professor and Course in the input list, no statements 
are generated as these classes have already been 
visited. This completes the process of generating 
statements for all classes relevant to the query.

Finally, the algorithm iterates through the data 
properties in the unmodified input list (name and 
courseName in this example). Since name can be 
associated with multiple classes i.e., GradStudent 
and Professor, hence the algorithm assigns different 
query variables to name, resulting in statements 3 and 4 
in Figure 3. The domain of data property courseName 
is Course, which leads to the generation of statement 
5. In the final step, after applying filters based on non-
empty values from the list of key-value pairs, the final 
query is formulated as shown in Figure 4, which is 
similar to the manually crafted query shown in Figure 1b.

3.4. Complexity Analysis

The first step in ASQFor is to find the smallest subgraph 
that connects all nodes relevant to the user provided 
concepts. The complexity of this step depends 
on the structure of the Ontology. Therefore, we analyze
the complexity of query generation in the special case of a tree Ontology.

Let \( k \) be the keywords in the user query and \( n \) be the total number of nodes in the Ontology. ASQFor traverses the path from a node corresponding to a keyword towards the root \( r \) of the Ontology \( G \), for each keyword, in order to compute the lowest common ancestor of all keyword mappings in the Ontology. This step requires \( O(k \log n) \) operations in the worst case, i.e., when each node corresponding to a user provided keyword lies on a separate branch of the tree. Once subgraph \( Q \) is constructed, ASQFor traverses \( Q \) to generate the SPARQL statements that constitute the query. It is easy to show that this step also requires \( O(k \log n) \) operations in the worst case. Therefore, the overall complexity of ASQFor is \( O(k \log n) \).

3.5. Limitations

To achieve the goal of providing non-expert users a subset of functionality of SPARQL while keeping our approach completely dynamic and domain independent, we have made certain decisions in designing ASQFor, which lead to the following limitations:

1. Currently ASQFor supports hierarchical Ontologies—it does not support cycles and self-properties. Hierarchical Ontologies are commonly used in multiple domains for cataloging or organizing information [4, 14, 18]. The reason behind this design choice is due to the complexity associated with finding a query subgraph (tree) that spans over selected query nodes in an arbitrary graph; this problem is known to be NP-hard [39].

2. For ASQFor to be able to issue queries over a semantic repository, a well-defined schema ontology must be available, i.e., the schema ontology must contain complete information about domains and ranges of all object properties, domains of all data properties, and complete hierarchy of subclasses. Since, ASQFor traverses the schema ontology during query formulation phase, missing information can result in formulation of incorrect query.

3. For ASQFor, we assume that the user query input is in the form of key-value pairs which exactly match corresponding ontological terms in the repository. Publicly available tools such as Stanford Parser [6] can be used to tokenize Natural Language sentences, followed by string matching techniques (e.g. [15]) to match tokens to ontological terms. However, this is beyond the scope of this work.

4. Aggregation queries (e.g. COUNT, SUM) are not supported in the current version of ASQFor. The reason is that the simplistic key-value pair syntax we chose to accommodate user query needs cannot be mapped to the entire SPARQL syntax. Secondly, functions such as COUNT, SUM and AVERAGE only affect the SELECT clause of the SPARQL query with possible addition of statements at the end of the query such as GROUP BY clause. The triple patterns in the body of the SPARQL query still rely on computing query subgraph that links all the ontological concepts relevant to the user query. Allowing user to specify the aggregating functions requires modification of how input is provided by the user. The algorithm to traverse the schema ontology will remain unchanged.

4. Experimental Evaluation

We evaluate ASQFor in two ways: (i) qualitatively to identify the possible ease-of-use and increased productivity in information searching activities as a direct result of reducing the amount of time spend to manually develop and adapt queries, and (ii) quantitatively to indicate possible performance overheads.

For practical reasons we have not measured productivity increase directly. Such direct evaluation would require measuring (i) query construction speed of users with and without ASQFor, and (ii) number of query tasks completed over a duration of time by users with
Table 2
Query Formulation for Representative Queries

<table>
<thead>
<tr>
<th>Queries</th>
<th>Q2</th>
<th>Q3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>\textless &quot;Industry&quot;, &quot;&quot; \textgreater , \textless &quot;Occupation&quot;, &quot;&quot; \textgreater</td>
<td>\textless &quot;Name&quot;, &quot;&quot; \textgreater , \textless &quot;School&quot;, &quot;3&quot; \textgreater</td>
</tr>
</tbody>
</table>

and without ASQFor. We measured instead the syntactic difference between queries generated by ASQFor as compared to optimized queries that were handcrafted by qualified programmers for the same information need and difference in their execution times.

To best of our knowledge, the source code of the other automatic query formulation tools such as Ginseng [3], Querix [17] and Squall [12] is not available for a fair comparison. Such systems rely on a web interface, hence it is impossible to measure the exact query formulation and execution time required and compare against ASQFor.

For quantitative evaluation, we compared query execution with and without ASQFor on a triple store created using Apache Jena\textsuperscript{5}. We evaluated four queries, ranging from simple to complex queries (see Table 3) using a dataset of varying size (ranging from 20\textsuperscript{2} 200,000 triples). We use query formulation time and query execution time to quantify the efficiency of ASQFor in the task of automatic SPARQL query generation. We also measured overall time, the sum of query formulation and execution times. We compare the performance of ASQFor against manually defined and hand-crafted optimized queries.

We implemented ASQFor as a Java function that takes a list of key-value pairs as input and returns valid

\textsuperscript{5}https://jena.apache.org/
Table 3 Evaluation Queries

<table>
<thead>
<tr>
<th>$Q_1$</th>
<th>Name, birthplace, gender and marital status of all people on active military duty.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_2$</td>
<td>Occupations in different industries.</td>
</tr>
<tr>
<td>$Q_3$</td>
<td>Names of people who attended private school.</td>
</tr>
<tr>
<td>$Q_4$</td>
<td>All attributes for people born in California.</td>
</tr>
</tbody>
</table>

4.1. Dataset

For evaluation purposes, we used the 1990 US Census data\(^6\), which is provided in tabular format. The dataset contains 68 attributes for 2,458,285 individuals in total. For evaluation, we randomly sampled this dataset to create smaller datasets of different sizes (20, 200, 2,000, 20,000, and 200,000). Each query, from Table 3, was issued five times on each dataset. Based on the Census Ontology shown in Figure 5, we converted the tabular data into RDF to be used for the experiments.

4.2. Quality and Efficiency of ASQFor Generated Queries

The efficiency or execution time of automatically generated queries depends on identified concepts and their relationships, as well as the number of intermediate triples that are retrieved from each statement in the query. To demonstrate that ASQFor generates “good quality” queries, we show the difference between the manually written and automatically generated queries $Q_2$ and $Q_3$ in Table 2. Comparing the two queries for $Q_2$, it can be verified that the manually created and the automatically-generated queries are identical.

On the contrary, for $Q_3$, some difference is observed. While the manual query refers directly to the “private” school\(^7\) filter, the automatic query uses the FILTER function provided by SPARQL to reduce the result set to the requested information. This difference in using FILTER keyword can also be seen in automatically formulated query shown in Figure 4 and equivalent manually crafted query shown in Figure 1b. As a result, the manual query requests only triples that contain data referring to private schools, whereas the automatic query retrieves initially all triples based on the keywords. This larger result set is then filtered afterwards. Using FILTER function this way can have an impact on query response time due to larger intermediate result set. The manual query is well optimized to the specific task at hand, and therefore is expected to perform better than the automatically generated query in this scenario. However, there is no difference in the returned results for both queries.

4.3. Effect of Automation on Query Formulation Time

To evaluate the performance of ASQFor in formulating queries, we measured the time required to generate the representative queries shown in Table 3. These queries differ in number of nodes and attributes they query and the depth of the query subgraph.

We measured the overhead introduced by query formulation over query execution time using our datasets of different sizes (20, 200, 2,000, 20,000, and 200,000). Figure 6 shows, in logarithmic scale, the average formulation time as compared to query execution time. This suggests that the overhead of ASQFor for query formulation is constant, whereas execution time varies as a function of the size of the result set and the size of


\(^7\)Options for the attribute “School” are encoded in the dataset as integers. (0 = N/A, 1 = Not Attending, 2 = Public, 3 = Private)
the repository. In fact, query formulation time is significant as compared to query execution time only when the repository is substantially small (i.e., less than 2,000 entries). As expected, with increasing repository size, query execution time surpasses query formulation time. The mean and standard deviation of the ratio of formulation time to total time (formulation + execution) is shown in Figure 8. It can be seen that the formulation time on average takes ∼90% of the total time execution times) for a dataset of size 20, whereas it accounts for <25% of the total time for a dataset of size 200,000 entries. Therefore, query formulation time becomes insignificant for large-scale semantic repositories.

4.5. Semantic Search of 1990 US Census Data

In this section, we show how ASQFor can be incorporated into a semantic search system, using the 1990 US Census data. For this, we have implemented a user interface (Figure 9) that supports a combination of semantic search and exploration queries using ASQFor. From the user’s perspective, he only needs to know what kind of information is available in the database irrespective of how it is organized using Ontological concepts and their interrelationships specified using object and data properties. This has led to our minimalist design which allows users to pick and choose the concepts that are relevant for the query, specify filtering values, and get the desired result. After selecting the required concepts, users can click the Filter Options to specify filtering values for individual concepts or leave them blank. The filtering values can be entered concatenated with comparison operators e.g. <500. for range queries. After clicking Submit Query, the results are returned in CSV format. The selection and filter values in Figure 9 corresponds to a query asking for “People with more than 16 years of education who are employed and making more than $100,000/yr”. Unlike other visual query interfaces [18], our primary focus is to abstract the details of SPARQL and schema Ontology from the end user, providing him only the data attributes to choose from. Furthermore, this interface can be dynamically generated from a schema Ontology, resulting in a portable application that only requires access to the semantic repository and builds a functional-to-SPARQL query translator and a GUI on the fly.

5. Conclusion

While the Semantic Web promises data sharing and re-usability without boundaries, harnessing the rich semantic data provided by knowledge bases on the Web has proven difficult for those ordinary end-users who
are not necessarily familiar with Ontologies or semantic query languages. To ensure that precise answers can be delivered to user queries while at the same time retaining the simplicity of keyword-based search, we have developed a framework that can answer complex semantic queries over structured repositories through a simple interface. We showed that our approach enables end-users to easily issue semantic queries that match their information needs without intimate knowledge of the underlying data representation or Semantic Web technologies. The run time of ASQFor depends only on the number of classes and attributes in the semantic database and not on the number of records in the database. As a result, the query formulation time remains constant despite the size of the semantic repository. Evaluation showed that ASQFor is efficient, even for large datasets.

The approach detailed in this paper is not one of yet another library but rather how a simple abstraction can be used to leverage the advantages of Semantic Web technologies while at the same time offering a convenient approach to end-users to access semantically rich data from a knowledge base through simple API. In this context, our ASQFor framework:

- adds a layer of abstraction between user and the SPARQL end point,
- does not rely on pre-defined templates, rules or dictionaries,
- has low query formulation overhead,
- queries the semantic repository to extract the relevant schema information (in the background),
- automatically generates SPARQL query based on user provided keywords.

5.1. Future Work

Our approach is generic, easy to use, and can be easily adapted to other domains by simply exchanging the data and the Ontology describing it as long as the Ontology describing the domain follows a certain structure. Currently ASQFor supports tree-structured Ontologies – it doesn’t support cycles and self-properties. Finding optimal query subgraph based on a given subset of nodes in an arbitrary graph is modeled as Steiner Tree problem in literature, which is NP-hard [39]. For this work, we have made this trade-off to provide at least a subset of the Semantic Web functionality to the end-users through a simple and intuitive key-value interface. We plan to eliminate this constraint in future work.

Finally, ASQFor doesn’t currently accommodate aggregation queries as it is not possible to map the entire SPARQL syntax into a key-value pair syntax. Even though this limitation can be addressed by introducing a specialized vocabulary in our method, we wanted it to be as generic as possible. We plan on incorporating other SPARQL functions in ASQFor in future work.
Acknowledgments

This work is supported by Chevron U.S.A. Inc. under the joint project, Center for Interactive Smart Oilfield Technologies (CiSoft), at the University of Southern California.

References


[28] M. Sander, U. Waltinger, M. Roshchin and T. Runkler, How Google is us-


cal Methods in Natural Language Processing and Computa
tional Natural Language Learning, Association for Computa

data, Melbourne, Victoria, Australia, May 31 - June 4, 2015, 2015, pp. 1809–1824.