ASQFor: Automatic SPARQL Query Formulation for the Non-Expert

Muhammad Rizwan Saeed, Charalampos Chelmis, Viktor Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California, Los Angeles, USA
{saeedm, chelmis, 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 and require expensive customization. Natural Language interfaces are particularly preferable to other interfaces for providing users with access to data, however the inherent difficulty in mapping NLP queries to semantic data is the ambiguity of natural language. 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 repository 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 the construction of query formulation while at the same time matching the precision and recall of hand-crafted optimized queries.

Introduction

As more and more semantic data become available on the Web, 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 SPARQL\(^1\). Despite their strong expressive power, such formal languages impose an initial barrier to adoption due to their hard requirement for knowledge of their formal syntax and understanding of the way knowledge is encoded in semantic repositories.

The Resource Description Framework (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 \(<\text{subject, predicate, object}>\), and 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\(^2\). Although SPARQL is a standard way to access RDF data, it remains tedious and difficult for end-users because of the complexity of the SPARQL syntax and the RDF schema (Sander et al. 2014). Let us consider a running example shown in Figure 1. For the natural language question “What are the names of the graduate students enrolled in CS570 course?”, 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 (Kaufmann and Bernstein 2010; Lopez et al. 2013). 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 (Kaufmann and Bernstein 2010; Wang et al. 2007; Damljanovic, Agatonovic, and Cunningham 2012; Schwitter and Tilbrook 2004; Unger et al. 2012; Waltinger et al. 2013; Lehmann and Bühmann 2011), Semantic Web Technologies (Kaufmann and Bernstein 2010; Yahya et al. 2012; Ngonza Ngomo et al. 2013), and visualization environments (Stewart 2014; Sander et al. 2014; Kaufmann and Bernstein 2010).

Modern query languages for the Semantic Web do not really support the handling of natural language text, requiring specialized solutions ranging from predefined templates...
ASQFor’s simple and intuitive tuple-based interface accepts semantic querying over a knowledge base represented by RDF. That requires virtually no end-user training to facilitate semi-automatic SPARQL Query Formulation (ASQFor) framework passed to functions using programming languages. Our approach makes SPARQL queries more straightforward, they (i) require expensive customization to each new domain or Ontology, (ii) adding new templates requires the involvement of domain experts and language engineers. Furthermore, Natural Language interfaces are (i) limited by ambiguity and (ii) 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 despite the fact that most queries consist only of few words. In fact, search engines have become popular because of their simplistic conceptual model, i.e., results include those documents that match the specified keywords. Unlike natural language interfaces, for which users quickly develop negative expectations due to the difficulties associated with parsing and interpreting natural language (Kaufmann and Bernstein 2010; Yahya et al. 2012; Ngonga Ngomo et al. 2013), concept-based queries can be used to capture the information needs of a user (e.g., “Graduate Students CS570”) while at the same time offering a Google-like search interface to the end-user. However, existing approaches (Lei, Uren, and Motta 2006; Zhou et al. 2007; Wang et al. 2008; Tablan et al. 2015) ignore relations between concepts, which are important for accurate SPARQL queries to be formulated.

In this paper, we take a <key, value> approach to the problem of querying a semantic data repository (e.g., the equivalent of the keyword-based query for Figure 1 would be “<Name, "">, <GradStudent, "">, <courseName, CS570;">”), which is similar to the way arguments are passed to functions using programming languages. Our Automatic SPARQL Query Formulation (ASQFor) framework is a reusable and extendable, domain independent approach that requires virtually no end-user training to facilitate semantic querying over a knowledge base represented by RDF. 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. Ambiguities in the queries (e.g., “Name” may refer to a “Student” or a “Professor” in our running example) are lazily resolved at the query formulation stage (as opposed to the query understanding phase in existing approaches) which can result in significant speedup of the query formulation stage.

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 spent to manually develop and adapt queries.

**Related Work**

Part of the Semantic Web vision is to provide Web-scale access to semantically annotated content (Berners-Lee, Hendler, and Lassila 2001; Cardoso 2007). This implies understanding users’ information needs accurately enough to allow for retrieving a precise answer using semantic technologies. For this reason, NLP and Ontology-based understanding of natural language for translation of end-user queries into formal queries have been explored (Sander et al. 2014). Compared to keyword-based search, systems based on natural language can imply semantic relationships between keywords using a whole sentence (Freitas et al. 2012). However, simple search-box and concept-based search interfaces have been shown to achieve comparable results to NLP query approaches (Duke, Glover, and Davies 2007). Additionally, 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 (Ferré 2014).

Since a user query does not need to be syntactically correct but must contain a minimum set of “relevant” concepts, we propose a keyword-based interface to the Semantic Web. This eliminates the need of pre-processing natural language phrases into discernible tokens before matching such keywords to concepts and attributes in the Ontology. Predefined templates (Höffner et al. 2013; Zheng et al. 2015) are also not necessary to query Ontologies.

Other approaches that avoid the challenges of natural language processing rely on controlled environments that guide
the user step by step with suggestions of terms that are connected in the Ontology (Bernstein, Kaufmann, and Kaiser 2005; Kaufmann, Bernstein, and Zumstein 2006) allowing meaningful queries to be formulated. Such semi-automated approaches however limit the range of queries end-users can issue against the semantic repository. Similarly, form-based query construction methods (Ferré 2014), require users to fill out a variety of information in web forms, which may be both cumbersome and time consuming. However, it is unclear what performance overhead such methods entail in query formulation time as well as their effect on query execution time. Finally, approaches that rely on predefined rules lack the flexibility of introducing new queries and require human intervention to formulate and perform target-oriented search tasks (Sander et al. 2014).

More recently, learning mechanisms that improve in response to the vocabulary used by the users have also been proposed in an attempt to balance domain customization and performance (Lopez et al. 2007). However such systems rely heavily on language processing and depend on a controlled vocabulary. 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 rather than a single edge. Additionally, instead of examining all paths between all pairs of vertices or imposing path length constraints, we construct SPARQL queries using a compact subgraph that covers relevant vertices.

Automatic SPARQL Query Formulation

The main goal of ASQFor is to enable end-users to formulate semantic queries over structured data in terms of classes and properties while being oblivious to the actual structure of the data. There are two key challenges in this problem: (i) how to represent the query intention in a structural way, and (ii) how to address the ambiguity of terms in user generated queries. Specifically, the underlying semantic data repository is a graph database, i.e., a triple store, therefore enabling query processing, we need a graph representation of the user provided input. Secondly, terms in the query may refer to different entities. For example, “Name” may refer to a “Student” or a “Professor” in our running example. We need to know which one is relevant to the user’s information need.

In order to address these challenges, we generate SPARQL queries in three steps. First, we map the user provided keywords to concepts and attributes in the Ontology. Our triples-based interface is populated by an Ontology, exposing the users to predicates they are familiar with while hiding technical details such as the database schema or the knowledge representation in the form of Ontology. The Ontology can be written in some formal language, e.g., the Web Ontology Language OWL2, which is standardized by the W3C. For each keyword with a non-empty value (e.g., <courseName, CS570> in our running example) we generate a filtering SPARQL statement that associates the keyword to the value in the form of a data property (e.g., ?course university : courseName “CS570”). The variable varDictionary stores query variables (to be used in formulating SPARQL query) corresponding to classes and data properties in the Ontology. This can also be used to keeps track of subsumption relationships in the Ontology. For instance, if attributes specific to a class and one or more of its super classes are relevant to the query, then the query variable corresponding to that class is used for generating query statements for its super classes. This is illustrated in Figure 1 in our running example. The variable ?gradstudent is used to generate the SPARQL statement that links it to university : name which is an attribute of class Person, the super class of GradStudent.

Second, we extract the semantic relations between concepts in the query, based on which, we build a semantic query graph Q to model the query intention. Each vertex u in graph Q is associated with an argument (i.e., key) and each edge e_{uv} represents the relation between concepts u and v. A relationship between two arguments may be simple or complex, i.e., represented by a path of length greater than 1. A simple relationship is a triple <u, relation, v>, whereas a complex relationship might involve multiple triples with intermediate variables. In our running example, <?gradstudent university : takesCourse ?course > represents the semantic relationship takesCourse between concepts gradstudent and course. To construct the semantic query graph Q, we compute the lowest common ancestor r of all the vertices in the query. This step is necessary to establish the smallest set of relationships between concepts and attributes in the query that lie on different branches of the RDF graph, such as attributes name and courseName and classes GradStudent and Professor in our running example.

We then construct the subgraph that connects all nodes u to the root r of the query subgraph Q by tracing the path from each vertex u to r. The SPARQL statement is being generated while traversing the subgraph to r by populating statements that correspond to semantic relations and intermediate nodes at each step. Specifically, ASQFor iterates through the list of unvisited class nodes, one at a time, marking each visited node at every iteration and classifying the current node as range of a user-defined object property, subclass of another class, or both. In the first and third cases, ASQFor traces the path towards r using the domain of the user-defined object property, and generates corresponding SPARQL statements using the current node, the object property of which it is range and domain of that object property. When the current node is a subclass of another class, the query variable assigned to the current node is also assigned to its superclass. The process is repeated until r or a visited node is reached for a given keyword, after which the next keyword is selected and the process is repeated until all keywords are examined.

Finally, the SPARQL query is executed on the semantic repository and results are returned to the user. The pseudocode for ASQFor is shown in Algorithm 1.

---

1 http://www.w3.org/TR/owl-overview/
Algorithm 1 ASQFor

**Input:** list $L$ of key value pairs $< k, V >$

**Output:** SPAQRL query 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. $Q, varDictionary ← \emptyset$
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 $domain(k)$ in $varDictionary$
6. end if
7. end for
8. $r ←$ lowest common ancestor of all $k \in L$
9. for each key-value pair $< k, v >\in L$ do
10. $currentNode ← k$
11. while ($currentNode.visited == 0$ and $currentNode ≠ r$) do
12. $currentNode.visited = 1$
13. if $∃$ triple $< domain, prop, currentNode >$ then
14. $Q ←$ insert triple $< varDictionary.get(domain), prop, currentNode >$
15. else
16. if $∃$ triple $< currentNode, rdfs : subClassOf, domain >$ then
17. $childVar ← varDictionary.get(currentNode)$
18. insert (or replace) pair $(domain, childVar)$ in $varDictionary$
19. end if
20. end if
21. $currentNode ← domain$
22. end while
23. if $k$ is a data property and $v ≠ \emptyset$ then
24. $Q ←$ insert filter statement for $k$ using $v$
25. end if
26. end for
27. return $Q$

**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 of the Ontology, 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 $O(k \log n)$.

**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 qualified programmers with and without ASQFor, and (ii) number of query tasks completed over a duration of time from end-users with and without ASQFor. Since such evaluation can be tedious and impractical in practice, we measured instead the syntactic difference between queries generated by ASQFor as compared to optimized queries that were hand-crafted by qualified programmers for the same information need and difference in their execution times.

To best of our knowledge, the source code of other automatic query formulation tools such as Ginseng (Bernstein, Kaufmann, and Kaiser 2005), Querix (Kaufmann, Bernstein, and Zumstein 2006) and Squall (Ferré 2014) 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 by such systems and compare against ASQFor.

For quantitative evaluation, we compared query execution with and without ASQFor on an Apache Jena repository. We evaluated four queries (ranging from simple to complex queries, see Table 1) using a dataset of varying size (ranging from 20 – 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 SPARQL query as a String. Each SPARQL query was then evaluated against the semantic repository using the Jena API. The results were also returned in JSON format. Each test was repeated 5 times. The tests were run on a standard Lenovo Ideapad laptop with dual core 2.20 GHz Intel Core i7 CPU and 8GB of RAM.

**Dataset**

For evaluation purposes, we used the 1990 US Census data\(^4\), which is provided in tabular format. The dataset contains 68 attributes for 2,458,285 individuals in total. For evaluation, we randomly sampled the dataset, selecting 1,000,000

---

\(^4\)https://jena.apache.org/

entries and 20 attributes for each record. We divided this dataset into 5 independent sets of size 200,000 triples each. Figure 2 shows the Ontology we used for our experiments. Based on this Ontology, we converted the tabular data into RDF triples in a prepossessing step. The triples and Ontology were stored in a Jena triple store.

**ASQFor Generated Queries Quality and Efficiency**

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. Query efficiency therefore translates directly to the “quality” of SPARQL statements and can be directly measured in the amount of requested and returned triples.

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 $Q_2$, no major difference can be observed. In fact, the manual and the automatically generated queries are identical. In contrary, for $Q_3$, a slight difference is observed. While the manual query refers directly to the “private” school filter, the automatic query uses the FILTER function provided by SPARQL to reduce the result set to match the information need of the end-user. As a result, the manual query requests only triples that contain data referring to private schools, whereas the automatic query retrieves initially all triples of type School. The larger result set is filtered afterwards, however, the automatically generated query is not optimal and is expected to have an impact on query response time due to a larger returned and processed result set. Even though the manual query is well optimized to the specific task at hand, and therefore is expected to perform better than the automatically generated query. Adding the filtering function may not be optimal, but considering the trade-off between performance and generalization, this simple solution works best.

**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 1. 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 datasets of different sizes (20, 200, 2,000, 20,000 and 200,000). Figure 3 shows in logarithmic scale average formulation time as compared to query execution time. We computed average time over the 5 sets in our dataset for each of the four queries in Table 1. Figure 3 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 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 5. 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.

**Effect of Automation on Query Execution Time**

Figure 4 shows the average response time calculated over the 5 sets in our dataset for each of the four queries in Table 1. For most queries the difference between the execution time of manual and automatic queries is insignificant for practical purposes. In fact, ASQFor adds only little overhead as compared to the manually optimized queries, particularly as the size of the dataset increases. On some queries (e.g., $Q_3$) ASQFor seems to match the run time of the manual query even for smaller datasets.

<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>

*Options for the attribute “School” are encoded in the dataset as integers. (0 = N/A, 1 = Not Attending, 2 = Public, 3 = Private)
## Table 2: Query Formulation for Representative Queries

<table>
<thead>
<tr>
<th>Queries</th>
<th>Q₂</th>
<th>Q₃</th>
</tr>
</thead>
</table>
| **Manual**    | SELECT DISTINCT ?industry
   ?occupation WHERE {
   ?workinfo census:Occupation
   ?occupation.} | SELECT DISTINCT ?name ?school WHERE {
   ?person census:hasEducation>
   ?eduinfo.
   ?person census:Name> ?name.
   ?eduinfo census:School> "3".} |
| **ASQFor Generated** | SELECT DISTINCT ?industry
   ?occupation WHERE {
   ?workinfo census:Occupation
   ?occupation.} | SELECT DISTINCT ?name ?school WHERE {
   ?person census:hasEducation>
   ?eduinfo.
   ?person census:Name> ?name.
   FILTER ( ?school = "3" )} |

### Figure 3: Comparison between query formulation and total query execution time using ASQFor

(a) Q₁  
(b) Q₂  
(c) Q₃  
(d) Q₄

### Figure 4: Comparison between execution times of ASQFor generated and manual queries

(a) Q₁  
(b) Q₂  
(c) Q₃  
(d) Q₄

## Conclusion and Future Work

While the Semantic Web promises data sharing and reusability without boundaries, harnessing the rich semantic data provided by knowledge bases in the Web has proven difficult for those ordinary end-users who are not necessarily familiar with domain specific semantic data, Ontologies, or SQL-like 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 engines, we 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 growth of the semantic repository. Evaluation showed that ASQFor is efficient, even for large datasets.

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 tree structure. 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.

## Acknowledgement

This work is supported by [Anonymized Sponsor].
References


