SPARQLIt: Interactive SPARQL Query Refinement

Yael Amsterdamer
Bar-Ilan University

Yehuda Callen
Bar-Ilan University

Abstract

We propose to demonstrate an interactive system called SPARQLIt, assisting users in the formulation of SPARQL queries. In SPARQLIt, users start by specifying a well-structured SPARQL query; the challenge is that the query may not be grounded to a specific ontology schema or content. The space of possible grounded queries to which the user query may be mapped is then well-defined, and SPARQLIt allows the user to interactively explore this space in a systematic way. A key component of this exploration is a pruning mechanism that is based on identifying query parts that are correct/incorrect, and using them to efficiently filter candidate queries of a certain form. Our demonstration will use the Yago Ontology to exemplify the ease of constructing queries with SPARQLIt without prior knowledge of its contents. We will allow participants to specify queries of their choice and will walk them through the interactive process of query refinement.

PREPRINT

1 Introduction

The widespread adoption of knowledge graphs as means for representing information calls for effective ways to allow users to query and explore them. SPARQL, the predominant query language for RDF graphs, allows specifying complex data selection criteria. Yet, even users who are familiar with the SPARQL syntax may find it difficult to exercise their knowledge in practice since the ontology contents and schema are typically highly complex and may contain many sparse relationships, synonymous categories, incomplete data and other irregularities.

These challenges are well-known and have been extensively studied, leading to different tools for SPARQL query formulation, including natural language interfaces (e.g., [1]), auto-complete suggestions (e.g., [2, 3]), query-correction proposals (e.g., [2]) and query-by-example tools (e.g., [4]). Despite this great progress, the problem is far from being solved. Specifically, due to the schema complexity, users are likely to be offered queries that do not match their intention, or that return no results. When this happens, the users have to manually try to either change their input (e.g., the text in natural language) or their interaction with the system (e.g., choose other auto-completions for some query elements), without a concrete idea of what lead to the problem and whether their changes are likely to solve it.

Header — Interaction count: 4

```
Select * {
?x lives_in india .
columbia has_graduate ?x
}
```

```
Select * {
?x <livesIn> ?a .
?x <graduatedFrom> <Columbia_University> .
?a <isLocatedIn> <India>
}
```

	original_element	proposed_element	assignment_example
☑	?x	?x	<Shanthi_Pavan>
☑	lives_in	<livesIn>	
☑		?a	<Chennai>
☑	has_graduate	<graduatedFrom>	
☑	columbia	<Columbia_University>	

Figure 1: SPARQLIt User Interface

To this end, we present a *novel interactive framework for assisting users in formulating SPARQL Queries*. The user starts by writing a "semi-formal" query that follows SPARQL syntax but is not grounded to the ontology that is being queried. Namely, entity and relation names as well as the query structure may be imprecise. In a sense, this may be considered as a sort of an intermediate approach between informal query specification (e.g., in natural language), and standard formal query engines. Recent empirical evidence [5] (in a relational context) indicates that such semi-formal queries are much easier to specify than formal ones, yet are considerably simpler to translate to a formal language in comparison to unstructured input such as plain text. We focus in this work on simple SPARQL selection queries, containing only variables and element names. This fragment is in fact the one that requires knowledge of the schema, and can serve as a basis for other query fragments (e.g., group-by and difference).

Next, SPARQLIt iteratively presents the user with proposals for formal, grounded queries along with output examples. Since parts of the proposed query correspond to parts of the semi-formal query, the UI then allows the user to accept or reject each part of the query and output example. Through this feedback, the system efficiently prunes candidate queries and gradually computes improved query proposals.

Consider for example a data analyst who is interested in searching for Indian residents who studied at Columbia University. KBs such as Yago [6] contain such information and may be queried to retrieve it. However, the correct formulation of the intended query depends on *how* the relevant information (who is Indian, at which university did one study) is captured in the queried KB. Assume the analyst is not familiar with the ontology specifics and has constructed the SPARQL query appearing at the top-left pane of Figure 1 (ignore, for now, the rest of the figure). Briefly, this query searches for assignments of some entity e in the KB to a variable `?x` such that the KB contains two facts about e: e lives in India, and Columbia lists e as one of its graduates. This query could

be incorrect with respect to the queried KB, e.g., for the following reasons: (1) The entity relevant to an input element appears in the KB under a different name, e.g., <Columbia_University> rather than columbia; (2) for a given input element there are several reasonable candidate matches, e.g., is the KB predicate <isCitizenOf> a good match for lives_in? (3) the query structure does not conform to the KB schema. E.g., no has_graduate predicate between universities and people exists in the KB, and instead the KB has an inverse relation called <graduatedFrom> between people and universities.

We define a set of query refinement operators such that applying an operator sequence to a semi-formal query yields a formal one. Examples for the operators include (1) grounding: renaming of the abstract entity and relation names occurring in the semi-formal query to concrete entity and relation names occurring in the KB (2) reordering: changing the order in which query element appear, e.g., exchanging the person and university elements in the above example; and (3) decomposition: splitting a triplet into multiple ones. This resembles transformations in query reformulation [7] and relaxation [8], but the initial input is not formally specified, and hence our transformations operate under syntactic similarity rather than under logical equivalence.

Continuing our example, one possible query yielded by such sequence of operators is shown on the top-right pane of Figure 1. Here, *grounding* was required for all the elements of the input, transforming, e.g., columbia into <Columbia_University>, *reordering* has replaced has_graduate with the inverse <graduatedFrom> and *decomposition* has transformed the lives_in relation into the combination of <livesIn> and <isLocatedIn> (to select places in India).

Naturally, the space of formal queries thereby obtained is very large even if we impose natural restrictions on the length of the operator sequence. As evident from the above example, determining which query in this space matches the user intention may require the user's judgement, which in turn should be frugally asked for. We employ two mechanisms to address this. The first is ordering the space using a ranking function, which we tailor based on the distance of each query from the original semi-formal query. Here, an advantage of our semi-formal input is that we can combine well-defined structural distance as in works on query relaxation (e.g., [8]), with per-element textual distance as in natural language query interfaces (e.g., [1]).

The second mechanism uses feedback from previous phases to efficiently prune the space of candidate queries. The bottom pane of SPARQLIt's UI in Figure 1 shows mappings between parts of the original query and of the proposed query, as well as a sample assignment to each variable. The user can accept or reject each part of the proposed query or the sample result by marking the relevant check-boxes. This feedback is then applied to filter candidate queries. In particular, by monotonicity, we can avoid considering all the candidate queries sharing a query part that does not conform to the user feedback. For example, if the user required <Shanthi_Pavan> to appear in the query output (by marking "V" in the top-most check-box in Figure 1), and the query part ?x <livesIn> <India> does not return this result, the system avoids the many query candidates that include this query part. The two mechanisms are used together, to compute candidate queries in order of ascending distance from the original query and prune (or avoid computing) unsuitable candidates, until the next minimum-distance suitable candidate is found and proposed to the user.

PREPRINT

We use these components to build our prototype implementation for SPARQLIt. We propose an interactive demonstration of the system, walking participants through the query refinement process and demonstrating its usefulness for posing realistic queries over the Yago Knowledge Base.

Related Work There are many lines of work on algorithms and systems that assist in query (re-)formulation for different languages, including SPARQL, and interactive solutions that allow for gradual refinement of the query. Among these are natural language or semi-formal interfaces to databases (e.g., [1, 9, 10]), query reformulation (e.g., [7]), query auto-completion (e.g., [2, 3]), semantic similarity search (e.g., [8]), keyword search over a data graph (e.g., [11, 12]), and query-by-example (e.g., [4, 13]). The unique features of our work is the combination of (1) our starting point being a semi-formal specification, an idea recently shown to be particularly effective for constructing queries of complex semantics [5], (2) our consequent notion of distance-based cost for candidate queries, (3) our pruning technique based on sub-queries, and (4) our particular form of interactive refinement that allows user feedback on *sample results and every query element*, increasing the means by which the user can steer the system to the correct query.

2 Technical Background

We next overview the key technical notions to be used in our system.

2.1 RDF and SPARQL Queries

Let $\mathcal{E}$ be a domain of entity names/literals (e.g., `India`, `University`). Let $\mathcal{P}$ be a domain of predicate names (e.g., `livesIn`). An *RDF dataset* is a set of triples modeled as a labeled directed multigraph $G = (V_G, E_G)$, where $V_G \subseteq \mathcal{E}$ is a finite set of vertices and $E_G \subseteq V_G \times \mathcal{P} \times V_G$ is a finite set of edges that connect entities/literals through predicates from $\mathcal{P}$. An edge is directed from the *subject* of the predicate to its *object*. We use *element* to uniformly refer to an entity, literal or predicate. We will sometimes represent edges/triples $\langle s, p, o\rangle, \langle s', p', o'\rangle$ by the *n3 notation* `s p o. s' p' o'`.

A *formal SPARQL query* is similarly defined as a multi-subgraph $Q = (V_Q, E_Q)$, with the exception that $V_G \subseteq \mathcal{E} \cup \text{Var}$ and $E_G \subseteq V_G \times (\mathcal{P} \cup \text{Var}) \times V_G$, i.e., names from $\mathcal{E}$ and $\mathcal{P}$ may be replaced with variables from Var, where it holds that $\mathcal{E} \cap \text{Var} = \mathcal{P} \cap \text{Var} = \emptyset$, and variable names start with `?`. The general SPARQL syntax is broader, and allows for filtering conditions, difference, and group by – but we focus here on simple selection as the most difficult part for a user unfamiliar with the KB contents. A *result* of $Q(G)$ of a formal query is a (multi-)subgraph $G' \subseteq G$ defined by a function $\varphi : \text{Var} \to V_G \cup E_G$ such that by replacing in Q every $x \in \text{Var}$ with $\varphi(x)$ we will obtain G', or, abusing notation, $G' = \varphi(Q)$. The *result set* of $Q(G)$ includes all such results (and may be empty).

We now introduce the notion of *semi-formal queries*, that have a similar form to that of formal SPARQL queries, but its labels are not bound to the corresponding set of names in the KB, i.e., $Q = (V_Q, E_Q)$, where $V_G \subseteq \mathcal{E}^u \cup \text{Var}$

PREPRINT

and $E_G \subseteq V_G \times (\mathcal{P}^u \cup \text{Var}) \times V_G$. We may assume that $\mathcal{E} \subseteq \mathcal{E}^u$, $\mathcal{P} \subseteq \mathcal{P}^u$ and $\mathcal{E}^u \cap \text{Var} = \mathcal{P}^u \cap \text{Var} = \emptyset$.

2.2 Interactive Refinement of Queries

Given a semi-formal query $Q = (V_Q, E_Q)$, we shall (iteratively) compute a formal query $Q_i = (V_i, E_i)$ and a partial mapping $\psi_i : V_Q \cup E_Q \to V_i \cup E_i \cup \{\perp\}$, where $\perp$ stands for the choice not to map some element of Q to Q_i. ψ_i will adhere to a set of constraints Γ_{i-1} derived from user feedback in iterations $0, \ldots, i-1$. We will aim to minimize a cost function Δ, that captures the dissimilarity of Q_i to Q. We will now define the notions of constraints and cost function, starting with the latter.

We view Q_i and ψ_i as the result of a sequence of edit operations on Q, where edits are split into *element-wise* and *structural* edits. Element-wise edits occur when ψ_i maps some $e \in \mathcal{E}^u \cup \mathcal{P}^u$ to a KB element in $\mathcal{E} \cup \mathcal{P}$, e.g., when the user-input `columbia` $\notin \mathcal{E}$ is mapped to `Columbia_University`$\in \mathcal{E}$. Structural edits are performed by edit operators that reorder, decompose or remove elements, such as the following.

- Object-Subject switching: given an edge $\langle s, p, o \rangle \in E_Q$ there is an edge $\langle s', p', o' \rangle \in E_i$ such that $\psi_i(s) = o'$ and $\psi_i(o) = s'$. E.g., map `India hasResident ?x` to `?x livesIn India`.
- Element to variable: given an edge $\langle s, p, o \rangle \in E_Q$, for some non-variable $e \in \{s, p, o\}$ ($e \notin \text{Var}$), we have the mapping $\psi_i(e) = v$ where v is a fresh variable in Var. E.g., map `?x resident India` to `?x ?p India`.
- Predicate splitting: given an edge $\langle s, p, o \rangle \in E_Q$, there are edges $\langle s', p', o' \rangle$, $\langle o', p'', o'' \rangle \in E_i$ such that $\psi_i(s) = s'$, $\psi_i(p) = p'$, $\psi_i(o) = o''$ and $o' \in \text{Var}$ – i.e., the subject and object are not connected directly but through a path of length 2, using variables for the intermediate elements. E.g., map `?x livesIn India` to `?x livesIn ?y. ?y isLocatedIn India`.

The cost Δ is then a weighted edit distance assigning each edit operation a cost that intuitively quantifies the semantic change incurred by the edit. We propose below a practical implementation of such a cost function.

Next, we allow users to provide a "yes"/"no" answer on each element mapping. A "yes" answer on $\psi_i(e) \notin \text{Var}$, means that this mapping will occur in every proposed query from Q_i and on. Conversely, a "no" answer will exclude such a mapping starting Q_{i+1}. To allow feedback on variables we present the user with a sample result $G' \in Q_i(G)$; in this case, a "yes" (resp., "no") answer for $\psi_i(e) \in \text{Var}$ will be interpreted as stating that the result set of any subsequent query Q_{i+k}, must (resp., cannot) include a result with the example assignment $\varphi(\psi_i(e))$. These accumulated constraints form a set Γ_i, which serves to narrow down the query search space.

2.3 Query Search Algorithm

As mentioned above, even for queries of bounded size the search space for formal queries is huge. We therefore employ an A^*-like search algorithm to traverse

PREPRINT

the possible queries in increasing order of cost, and further employ pruning to discard unsuitable queries, as follows.

For the cost function, we use a black-box search engine for elements, that may also account for linguistic aspects such as tokenization, semantic similarity, etc. Given an input element e, assume that e' is returned by the search engine with relevance $\mathrm{relev}(e, e')$. The cost of editing e into e' is the inverse of relevance, i.e., $\frac{1}{\mathrm{normalize}(\mathrm{relev}(e,e'))}$, using linear normalization to ensure relevance scores are in the range $[1, \infty)$. We associate a cost with each type of structural edit, defined to roughly capture the increase in the search space that they incur: e.g., subject-object switching has a higher cost than per-element edit. The cost of a proposed query is then the minimal sum of costs in a sequence of edit operations yielding it (i.e. its edit distance from the original semi-formal query).

For the search algorithm, we employ an A^*-like search in two levels: for structural edits, we start at the root with the structure of the input semi-formal query, and from each node branch out by considering every possible edit step. Some branches may merge, e.g., if we apply subject-object switch twice on the same element, or perform the same edits in a different order. For element-wise edits, we use the black-box search engine to retrieve a ranked list of candidates per input element, such that each combination of such candidates is a candidate grounding for the entire query. We start at the root by taking the top-1 candidate of each element and combining them; from each node we then branch out by considering every replacement of a candidate by the next-best candidate. The queries/structures with minimal cost are stored in frontiers, such that as soon as a query is rejected we add its branches to the frontier and proceed to the next minimal-cost query.

We introduce some optimizations to this general scheme. First, as a major optimization, we allow users to search only queries that return a non-empty result-set. This eliminates many candidate queries that are syntactically correct but have no realization in the KB. The form of queries that we consider are *monotone*, in the sense that adding triples/edges only reduces the number of results. Thus, we can specifically test for the emptiness of sub-queries against a triple store, cache the results, and eliminate search branches and candidate queries that include this sub-query. For efficiently returning initial results, we perform a triple-wise search using the search engine, which returns the most relevant triples (say, top-10000) for each triple in the input semi-formal query. E.g., for `?x lives_in india` it may return `Aadya livesIn India`, `Aarav livesIn India`, ...`Aaron livesIn Indiana`, ...`Diya citizenOf India` and so on. By intersecting the sets of assignments to the triple variables we may quickly find an assignment to the relevant variable (in this case, `?x`) that is consistent across triples and that uses the minimum cost element edits. Finally, if the results set of a sub-query is of small size (but non-empty), we extend them to larger sub-queries by examining all neighbouring edges of the sub-query results. This allows us to efficiently discover all possible groundings for additional parts of the query.

3 System Overview

SPARQLIt is implemented in .Net, using Blazor [14] for its front-end. Figure 2 depicts the main modules of SPARQLIt and their interplay. The user

PREPRINT

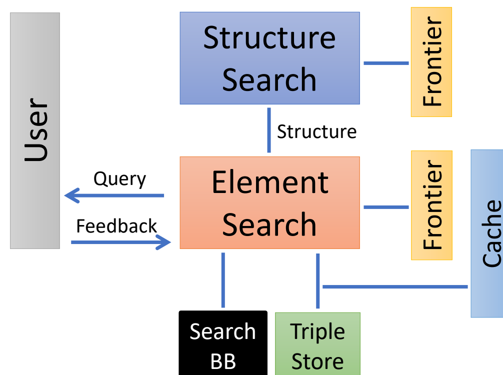

Figure 2: High-level Architecture

starts by specifying a semi-formal SPARQL Query via the User Interface as shown in the top-left corner of Figure 1. This triggers an interactive process of searching for the user-intended query. The process starts with the structure of the input (semi-formal) query and searches for a mapping of its individual elements to parts of the ontology. It interacts with a black-box search engine (we use Elasticsearch [15]), in which we treat the query elements and ontology triples as strings and look for approximate matches. The matches are ranked by cost and filtered (as explained in the previous section), and yield candidate queries. The resulting queries can be applied to the underlying triple-store to verify constraints on the output (we use Apache Jena [16]). We further store in a cache information on unsuitable sub-queries, which allows to avoid their materialization in other structures. Non-empty query results as well as the query and corresponding element mappings are presented to the user (see top-right of Figure 1 for an example query, and the bottom pane for mappings). The system then prompts the user for feedback (see check-boxes in the bottom pane of Figure 1). Accordingly and based on the cost function, the system either searches new element mappings for the same structure or invokes a structure search, i.e., examines structure edit operators. This process continues until the user is satisfied with the result or decides to restart the process, e.g., by submitting a new query.

PREPRINT

4 Demonstration

We will demonstrate SPARQLIt using the Yago ontology [6], focusing for the demonstration purposes on queries related to academic information (e.g. involving relations such as Doctoral advisor, university graduates, etc.), but connecting such facts to other domains, such as places around the globe, politics and sports. The demonstration will serve to show that, with SPARQLIt, one may formulate queries without familiarity with the queried ontology. We will first demonstrate some pre-selected queries, showing the interaction with our system in cases where different operators are needed to reach the correct formal query. We will show examples of positive and negative feedback on query parts and results, as well as the gradual traversal of the search space. We will then invite participants to write semi-formal SPARQL queries of their choice in this domain

and feed them into SPARQLIt. We refer the reader to the accompanying video for an example of the interactive process to be demonstrated.

For the second part of the demonstration, we will allow participants to look "under the hood" of the interactive process they have went through. We will in particular show candidate queries generated but pruned, the cost assigned to them, and other decisions made by the underlying search algorithm.

Acknowledgements

This work was funded in part by the Israel Science Foundation (grant No. 1157/16).

PREPRINT

References

[1] D. Diefenbach, K. D. Singh, and P. Maret, "WDAqua-core1: A question answering service for RDF knowledge bases," in *WWW comp.*, 2018.

[2] A. El-Roby, K. Ammar, A. Aboulnaga, and J. Lin, "Sapphire: Querying RDF data made simple," *PVLDB*, vol. 9, no. 13, 2016.

[3] K. Rafes, S. Abiteboul, S. C. Boulakia, and B. Rance, "Designing scientific SPARQL queries using autocompletion by snippets," in *eScience*, 2018.

[4] G. I. Diaz, M. Arenas, and M. Benedikt, "Sparqlbye: Querying RDF data by example," *PVLDB*, vol. 9, no. 13, pp. 1533–1536, 2016.

[5] T. Wolfson, M. Geva, A. Gupta, Y. Goldberg, M. Gardner, D. Deutch, and J. Berant, "Break It Down: A question understanding benchmark," *Trans. Assoc. Comput. Linguistics*, vol. 8, 2020.

[6] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: A Core of Semantic Knowledge," in *16th International Conference on the World Wide Web*, 2007.

[7] M. Benedikt, E. V. Kostylev, F. Mogavero, and E. Tsamoura, "Reformulating queries: Theory and practice," in *IJCAI*, 2017.

[8] W. Zheng, L. Zou, W. Peng, X. Yan, S. Song, and D. Zhao, "Semantic SPARQL similarity search over RDF knowledge graphs," *PVLDB*, vol. 9, no. 11, 2016.

[9] F. Li and H. V. Jagadish, "NaLIR: an interactive natural language interface for querying relational databases," in *SIGMOD*, 2014.

[10] F. Li, T. Pan, and H. V. Jagadish, "Schema-free SQL," in *SIGMOD*, 2014.

[11] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional expansion for keyword search on graph databases," in *VLDB*, 2005.

[12] K. Golenberg and Y. Sagiv, "A practically efficient algorithm for generating answers to keyword search over data graphs," in *ICDT*, 2016.

[13] K. Dimitriadou, O. Papaemmanouil, and Y. Diao, “Explore-by-example: an automatic query steering framework for interactive data exploration,” in *SIGMOD*, 2014.

[14] “Blazor,” https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor/.

[15] “Elasticsearch,” https://www.elastic.co/elasticsearch/.

[16] “Apache jena,” https://jena.apache.org/.

PREPRINT