Provenance-Based SPARQL Query Formulation

Yael Amsterdamer
Bar-Ilan University

Yehuda Callen
Bar-Ilan University

Abstract

We present in this paper a novel solution for assisting users in formulating SPARQL queries. The high-level idea is that users write "semi-formal SPARQL queries", namely, queries whose structure resembles SPARQL but are not necessarily grounded to the schema of the underlying Knowledge Base (KB) and require only basic familiarity with SPARQL. This means that the user-intended query over the KB may differ from the specified semi-formal query in its structure and query elements. We design a novel framework that systematically and gradually refines the query to obtain candidate formal queries that do match the KB. Crucially, we introduce a formal notion of provenance tracking this query refinement process, and use the tracked provenance to prompt the user for fine-grained feedback on parts of the candidate query, guiding our search. Experiments on a diverse query workload with respect to both DBpedia and YAGO show the usefulness of our approach.

1 Introduction

A huge body of information is stored in RDF knowledge Bases (KBs) such as YAGO [34], DBpedia [12] and others. Such KBs can be queried using SPARQL, the W3C standard query language [33]. Familiarity with the SPARQL syntax, however, is not sufficient for using it: writing a SPARQL query requires a deep understanding of the KB content and structure. In turn, in many useful KBs, this structure is highly complex and contains many irregularities. The KB structure often further evolves over time, and so even users who are initially familiar with it may struggle to adapt to such modifications.

To this end, our system SPARQLIt[1] (see Figure 1 for a high-level architecture) allows users to write a "semi-formal" query (step 0 in the Figure), i.e., a query whose syntax follows that of SPARQL, but its contents – entities and properties – do not necessarily match the KB. For example, a user seeking graduates of Columbia University living in India may write a query with the selection criteria `?x livesIn India. ?x graduatedFrom columbia`. When evaluated over YAGO, the user may discover that the query returns little or no results. The reason is that in YAGO, `livesIn` is usually populated with cities rather than countries and that Columbia is represented as `<Columbia_University>` – but it is difficult for a user to discover this. To this end, our solution generates candidate formal queries, i.e., queries that match the KB. Semi-formal queries are transformed to candidate formal queries via sequences of operations of two flavors: *structural edits* (step 1 in Figure 1), which are proposals for similar, alternative structures of semi-formal queries, e.g., replacing `?x livesIn India` by `?x`

[1] The code of SPARQLIt implementation is available at [11].

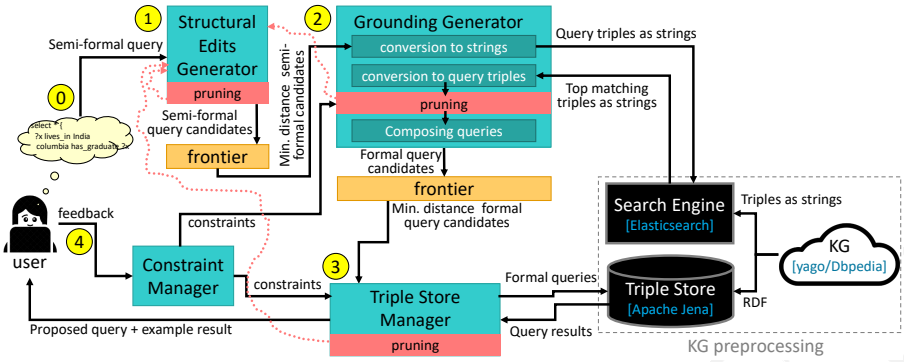


Figure 1: Overall architecture of SPARQLIt

`livesIn ?y. ?y isLocatedIn India`; and *groundings* (step 2) where elements of semi-formal queries are replaced by elements of the KB, e.g., replacing `columbia` by `<Columbia_University>`. We detail in Sections 3-5 the ways in which structural edits and groundings are generated to yield relevant candidate queries. Candidate queries are evaluated over the KB (step 3) so that query results may also serve to filter candidates (see below).

Crucially, we introduce a formal notion of *provenance tracking* throughout the process: we track the sequence of transformations that are performed, leading from the input semi-formal query to each candidate formal query, as well as the binding of variables of candidate queries to the KB. Provenance tracking serves as the basis of procuring *fine-grained feedback* from the user with respect to proposed formal queries. Namely, for a candidate formal query, we present not only the query itself and its example evaluation results, but also the *way* in which each element in the candidate query was obtained, i.e., from which element in the user's semi-formal query it has been (indirectly) transformed, if any. This is combined with the provenance of query evaluation (i.e. binding of formal query variables to KB elements). See bottom pane of Figure 2 for an example of presented provenance, in the form of (semi-formal element, formal element, binding example). Feedback is then procured with respect to each such piece of provenance: the user may mark "must", "must not" or "don't care". The user feedback is converted by the *Constraints Manager* (step 5) to constraints on proposed queries, which are accumulated and used by the other modules to prevent the generation of queries that do not comply with user feedback. This is repeated until the user finds a proposed query satisfactory.

We have implemented our solution in a prototype system (SPARQLIt) and examined (Section 6) its performance over two large-scale KBs, YAGO [34] and DBpedia [12], and a standard query benchmark [35]. The experimental results support the practicality of our approach, requiring only few interactions and a few seconds to find most of the examined queries.

2 Model

We start with preliminaries on RDF, then introduce our notions of semi-formal queries and provenance.

PREPRINT

RDF Knowledge Bases. An RDF Knowledge Base (KB) can be abstractly viewed as a set of facts in the form of triples. Let Ent be a domain of entity names (e.g., `India`, `University`) and Lit be a domain of literals (e.g., `2021`). Let Pred be a domain of predicate names (e.g., `livesIn`). An *RDF knowledge base* is a set of triples of the form (s, p, o) where $s \in \text{Ent}$ is the subject, $p \in \text{Pred}$ is the predicate and $o \in \text{Ent} \cup \text{Lit}$ is the object. We use *element* to uniformly refer to an entity, literal or predicate. We will sometimes represent multiple triples $\langle s, p, o\rangle, \langle s', p', o'\rangle$ by the *n3 notation* `s p o. s' p' o'`.

Formal SPARQL selection queries. To query RDF, we use the notion of Basic Graph Patterns (BGPs). Let $\text{Var} = \{\texttt{?x}, \texttt{?y}, \dots\}$ be a set of variables. A *Triple Pattern* is a member of the set $(\text{Ent} \cup \text{Var}) \times (\text{Pred} \cup \text{Var}) \times (\text{Lit} \cup \text{Pred} \cup \text{Var})$. Namely, in a triple pattern, subjects, predicates and objects may be replaced by variables. A BGP is a set of triple patterns, and its graph view may be obtained in the same way RDF KBs are encoded as graphs. A *formal SPARQL selection query* $Q = (G_Q, V_Q)$ then consists of a BGP G_Q and a set of output variables V_Q, which is a subset of the variables occurring in G_Q.

Query evaluation. Given a formal SPARQL selection query $Q = (G_Q, V_Q)$ and an RDF KB G, let φ be a mapping of all variables in G_Q to RDF terms in G. Denote by $\varphi(G_Q)$ the result of replacing in G_Q every variable v by $\varphi(v)$. If $\varphi(G_Q) \subseteq G$ (i.e. all obtained triples are in the KB G) then we say that φ is a *binding*. Each binding φ yields a *query answer* $A = \varphi|_{V_Q}$ (φ restricted to output variables) and the *query result* $Q(G)$ is the set of all such answers.

Semi-formal queries. We now introduce the notion of *semi-formal queries*, that have a similar form to that of formal SPARQL queries, but their labels are not necessarily bound to the corresponding set of names in the KG. Additionally, they may include special temporary place-holder elements from $\text{Temp}^{sf} = \{\texttt{??X}, \texttt{??Y}, \dots\}$ to be replaced by any KB term (entity/literal/predicate) when we transform the query into a formal one (see below). Formally, let $\text{Ent}^{sf} \supset \text{Ent}$, $\text{Lit}^{sf} \supset \text{Lit}$, $\text{Pred}^{sf} \supset \text{Pred}$ be extended sets of entity, literal and predicate names, abstractly capturing any element the user may write, including formal elements. A semi-formal BGP is then a BGP whose triple patterns are elements of $(\text{Ent}^{sf} \cup \text{Var} \cup \{\text{Temp}^{sf}\}) \times (\text{Pred}^{sf} \cup \text{Var} \cup \{\text{Temp}^{sf}\}) \times (\text{Ent}^{sf} \cup \text{Lit}^{sf} \cup \text{Var} \cup \{\text{Temp}^{sf}\})$. A *semi-formal query* $Q = (G_Q, V_Q)$ consists of a semi-formal BGP and a distinguished subset V_Q of output variables.

Example 2.1. *Figure 2 is a screenshot of SPARQLIt, where the top-left panel displays a semi-formal query: its syntax follows that of SPARQL, yet some of its elements, e.g.,* `columbia`, `has_graduate`, *do not occur in the queried KB (YAGO). The top-right panel displays a formal SPARQL query matching YAGO, for which an example binding is* $\varphi(?x) =$`Rajnesh_Domalpalli` *and* $\varphi(?s2) =$`Hyderabad`. *The binding* φ *yields the following YAGO triples:*
`Rajnesh_Domalpalli graduatedFrom Columbia_University. Rajnesh_Domalpalli livesIn Hyderabad. Hyderabad isLocatedIn India.`

Provenance model. We introduce two types of provenance. The first is "standard": the provenance of a binding φ obtained by evaluating a formal

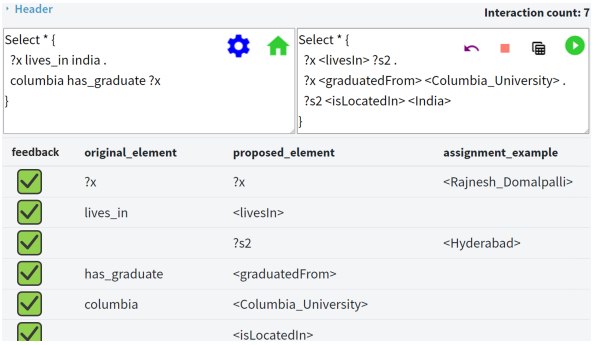


Figure 2: SPARQLIt User Interface

SPARQL Query Q over a KB G, denoted $\text{prov}(Q, G, \varphi)$, is represented as a set of variable-value pairs of the form (x, v). The provenance of a query answer $A \in Q(G)$, denoted $\text{prov}(Q, G, A)$ is then a set of such provenance representations, for all the bindings of Q in G yielding A. The second type of provenance is novel, and is geared towards tracking the gradual refinement of queries, as follows.

Definition 2.2. *Given two (formal or semi-formal) BGPs Q and Q', a* provenance expression *for a transformation of Q to Q' is denoted by* $\text{prov}(Q, Q') = (P, C)$*: P is a set of pairs (e, e') where e is either $\perp$ or an element of Q and e' is either $\perp$ or an element of Q', such that (1) $(\perp, \perp) \in P$ and (2) each element of Q and of Q' appears in exactly one pair. $C \in \mathtt{N}$ is the transformation cost.*

Intuitively, pairs encode "mappings" of individual elements in Q to elements in Q'; the notation $\perp$ is used to mark deletions/insertions of elements.

We will show in the sequel how to attach provenance to concrete transformations, yet we already note that an important property of provenance is composability: namely, need to be able to combine provenance expressions of a sequence of refinements to yield a provenance expression for the entire sequence.

Definition 2.3. *Let Q_0, Q_1, Q_2 be three queries, and let* $\text{prov}(Q_0, Q_1) = (P_0, C_0)$ *and* $\text{prov}(Q_1, Q_2) = (P_1, C_1)$*. We compose them to provenance* $\text{prov}(Q_0, Q_2) = (P_2, C_2)$ *by* $P_2 = \{(e_0, e_2) \mid \exists e_1 \neq \perp . (e_0, e_1) \in P_0 \wedge (e_1, e_2) \in P_1\} \cup \{(\perp, e) \mid (\perp, e) \in P_1\} \cup \{(e, \perp) \mid (e, \perp) \in P_0\}$*. As for cost, $C_2 = C_0 + C_1$.*

Intuitively, mappings are composed wherever elements occur in Q_1; otherwise, elements are either deleted in Q_1 or inserted only in Q_2, and the provenance records this insertion/deletion. The cost of transformations is cumulative.

3 Structural Edits

A first type of edits that we apply to semi-formal queries are geared towards modifying the query structure. Edits are applied to triple patterns, capturing reordering/deletion/insertion of the following flavors (we note, however, that our approach is generic and other types of edits may easily be incorporated). For each edit operation, we also define its provenance (see definition 2.2), but keep the costs abstract at this point and discuss concrete cost choices below.

Subject-Object Switching: switch the subject and object of $T = \langle s, p, o \rangle$, yielding $T' = \langle o, p, s \rangle$. The provenance captures an identity mapping, i.e., its set of pairs is $\{(e, e)\}$ for each element e of the (original and result) query.

PREPRINT

Element Exclusion: replace the subject/object/predicate of $T = \langle s, p, o \rangle$ by a fresh variable `?x`, yielding $T' = \langle \texttt{?x}, p, o \rangle$ or $T' = \langle s, \texttt{?x}, o \rangle$ or $T' = \langle s, p, \texttt{?x} \rangle$. The fresh variable is not in the output and thus can be bound to any element of the KB. For $T' = \langle \texttt{?x}, p, o \rangle$ the provenance will include the pairs $(s, \bot)$, $(\bot, \texttt{?x})$, (p, p) and (o, o) and (e, e) for every other element e; the provenance is similarly defined for $T' = \langle s, ?x, o \rangle$ or $T' = \langle s, p, ?x \rangle$. Note that the excluded element is mapped to $\bot$ which means we indeed stop tracking it.

Predicate Splitting: replace $T = \langle s, p, o \rangle$ by two triples $T' = \langle s, p, \texttt{?x} \rangle$, $T'' = \langle \texttt{?x}, \texttt{??Y}, o \rangle$ where `?x` is a fresh variable and `??Y` is a fresh placeholder. This stands for replacing a direct predicate by a path including two predicates. The provenance includes the pairs (s, s), (p, p), $(\bot, \texttt{?x})$, $(\bot, \texttt{??Y})$, (o, o), and (e, e) for every other element e.

Example 3.1. *Consider the triple pattern* `?x lives_In India`. *If an inverse predicate is used in the KB, a candidate query may be generated by Subject-Object Switching yielding* `India lives_In ?x`; *the provenance expression* (e, e) *for each element* e *connects the subject (object) of the original triple pattern to the object (subject) of the refined one. Alternatively, applying Element Exclusion could yield* `?x ?p India` *or* `?x lives_In ?y`, *generalizing the query so that instead of the particular relation or entity we place a variable that may be bound to any predicate/entity. The provenance includes a record of the newly added variable (*$(\bot, \{$`?p`$\})$ *or* $(\bot, \{$`?y`$\})$*) and associates the other nodes and edges with their counterparts in the refined query. Last, it may be the case that the KB includes information about people living in cities rather than directly in countries. Applying Predicate Splitting on* `?x lives_In India` *would yield the two triple patterns* `?x lives_In ?s2. ?s2 ??P1 India`. *The placeholder* `??P1`$\in \text{Temp}^{sf}$ *will be ultimately replaced in further edit steps by a predicate such as* `isLocatedIn`.

Example 3.2. *Reconsider the semi-formal query in Figure 2, and consider the application of Predicate Splitting to* `?x lives_In India` *to* `?x lives_In ?s2. ?s2 ??P1 India`, *followed by Subject-Object Switching for* `columbia has_graduate ?x`. *Composing the two provenance expressions we obtain the pairs* $(\bot,$ `?s2`$)$ *and* $(\bot,$ `??P1`$)$ *along with pairs* (e, e) *for* $e =$ `India`, `?x`, *etc.*

Searching for structural edits We next briefly describe the process of generating candidate queries. We store a *frontier* of semi-formal queries, initially including only the input one. Whenever prompted, the Generator applies to each semi-formal query currently in the frontier, the least costly possible edit. Two types of pruning are applied to the generation of semi-formal queries. First, multiple sequences of edit operations may result in the same semi-formal query, in which case we keep in the frontier only the minimum-cost representative. Second, the structural edits generator maintains a cache of semi-formal sub-queries for which no formal query exists, i.e., they were rejected by other modules (as described in the sequel). These and queries contained in them are ignored in subsequent steps. Queries that are not pruned are stored in the frontier and the minimal-cost candidate is passed on to the Grounding Generator, along with its provenance.

Cost. The assignment of cost for each operation may be viewed as a configuration choice. We have experimented with different cost assignments, and observed

that it generally useful to render a single structural edit operation more costly than grounding the entire query (i.e. we prefer groundings that use the current structure, if exist). We thus set the structural edit costs to be greater than C, which is an upper bound on the grounding cost (see Section 4). Specifically, we have superior optimal results with costs $2C$, $100C$ and $30C$ for Object-Subject Switching, Element Exclusion and Predicate Splitting respectively.

4 Grounding Generator

The Grounding Generator gets as input a semi-formal query Q' and the KB G. It generates formal queries by replacing entities/predicates in Q' that do not occur in G by ones that do. We start by generating a ranked list of groundings for individual triple patterns in Q and then combine groundings that are consistent with each other to form a query. We next explain each of the two steps.

Triple Groundings. Given a semi-formal triple pattern t' we generate a ranked list of k formal triple patterns $t_1, ..., t_k$. These are generated as follows. First, we represent t as a string $s(t)$ by removing SPARQL syntax (including variables and placeholders) and performing tokenization. Then we feed $s(t)$ to a *black-box search engine* that indexes triples from the KB. The engine returns the top-k relevant KB triples along with their string representation. We augment these triples back to triple patterns, plugging into them any variable that has occurred in t'. Unlike variables, placeholders are not added back to the triple patterns, so that they are grounded by KB elements.

Example 4.1. *Consider for example the triple pattern* `?x lives_in India`. *First, we remove SPARQL notations and perform an initial tokenization, which yields* "lives in India". *The search engine results includes, e.g., the strings* "Aadya lives in India" *and* "Aarav lives in indianapolis", *attached to the KB triples* `<Aadya> <livesIn> <India>`, `<Aarav> <livesIn> <Indianapolis>`. *Since the original triple pattern has* `?x` *as subject, we replace the subject in the KB triples by* `?x` *and obtain the ranked list* `?x <livesIn> <India>`, `?x <livesIn> <Indianapolis>`.

Provenance. We define the provenance for groundings in a similar way to that of refinements. For a semi-formal triple pattern $t' = (s', p', o')$ and a choice of formal triple pattern $t = (s, p, o)$ as grounding, we introduce the pairs $(s', s), (p', p), (o', o)$ to be stored in the provenance. We discuss costs below.

From grounded triple patterns to formal BGPs. We use the ranked lists of formal triple patterns obtained for each triple pattern t' in the semi-formal query, to yield candidate formal BGPs. We traverse these lists in order, each time choosing a single candidate triple pattern for each t' (i.e. we start with the set of all top-1 triple patterns). For each choice of triples, we check their provenance for consistency, namely, that no two pairs $(x, y), (x, z)$ such that $y \neq z$ appear in their provenance. If the set is consistent then the formal triple patterns are concatenated to form a BGP G_Q. Otherwise (or when the Grounding Generator is prompted for the next query), the process is repeated, with one of the triple patterns being replaced by the next-best one in the ranked list, and so forth.

PREPRINT

Provenance revisited. Recall that only triple patterns with consistent provenance expressions were combined. The overall provenance is then defined as follows: its pairs set is the union of pairs sets in the provenance of all triple patterns; the cost is the sum of costs stored in these provenance expressions.

From BGPs to queries. So far, we have generated formal BGPs as candidates, mapping the semi-formal BGP to each of them. Recall that the semi-formal query Q' includes a distinguished subset $V_{Q'}$ of output variables. For each formal BGP G_Q with provenance $\text{prov}(G_{Q'}, G_Q) = (P, C)$, the set of output variables is defined as $V_Q = \{v \mid (v', v) \in P \wedge v' \in V_{Q'}\}$. The query $Q = (G_Q, V_Q)$ is the obtained candidate, with the carried provenance staying intact, i.e., $\text{prov}(Q', Q) = \text{prov}(G_{Q'}, G_Q)$. We next show how provenance of edits and groundings may be composed.

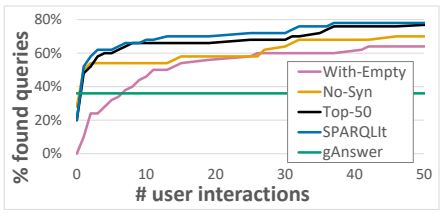
Example 4.2. *Following a sequence of structural edits as in Example 3.2, we generate candidate groundings for* `lives_In`, `India`, `has_graduate`, `columbia` *and the placeholder* `??P1`. *One such combination of groundings may be* `<livesIn>`, `<India>`, `<graduatedFrom>`, `Columbia_University`, *and* `isLocatedIn` *respectively. The resulting formal query is shown on the top-right part of Figure 2 and its provenance is shown on the bottom: e.g.,* `has_graduate` *has transformed to* `<graduatedFrom>`, *while* `isLocatedIn` *is newly added (so it has no original element counterpart).*

Cost. The cost for grounding a triple pattern t to t' is set based on the string distance measures between their representative strings $s(t), s(t')$, generated by the search engine as explained above. Specifically, we use the Levenshtein edit distance. An exception is the grounding of placeholders, which has 0 cost by definition, and is thus excluded from edit distance computation. To account for *semantic synonyms* that are represented by very different strings, we generate a set of synonyms for each string (using `https://www.datamuse.com/api/`), and define the distance as the minimal edit distance between a synonym of $s(t)$ and a synonym of $s(t')$. We revisit this design choice in Section 6. Last, recall that our choice of structural edit costs relied on an upper bound C for the grounding cost; we set C to be the maximal string length of a representative string of an element in the KB, multiplied by the number of query terms.

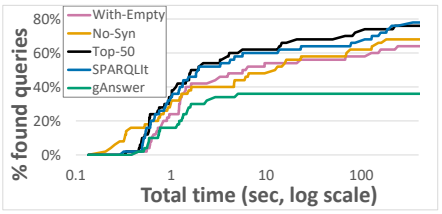
5 Procuring Feedback

The Triple Store Manager receives as input a formal SPARQL query Q produced by the Grounding Generator, and executes it over a black-box triple store (we have used Apache Jena [5]). The query result may be empty: this is typically an indication that Q does not match the user intention, and we search for alternative queries (we revisit this assumption in Section 6). If the query result is non-empty, we choose an example result, and an example binding yielding it. To procure feedback, we combine the provenance accumulated throughout the process of generating the candidate query, with the provenance of the example query result.

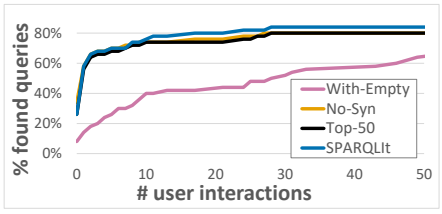
Definition 5.1. *Let Q', Q be a semi-formal and formal query respectively and let* $\text{prov}(Q', Q) = (P, C)$. *Further let G be a KB, $A \in Q(G)$ an answer,*



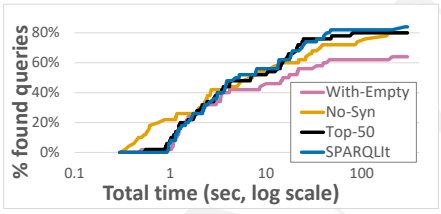
(a) Success by no. interactions (DBPedia)



(b) Success by total time (DBPedia)



(c) Success by no. interactions (YAGO)



(d) Success by total time (YAGO)

Figure 3: Overall performance for DBPedia and YAGO

φ *a binding yielding* A *and* $\text{prov}(Q, G, \varphi)$ *its provenance. The provenance* $\text{prov}(Q', Q, G, \varphi) = (P', C)$ *where* $P' = \{(e, e', v) \mid (e, e') \in P \wedge e' \in \text{Var} \wedge (e', v) \in \text{prov}(Q, G, \varphi)\} \cup \{(e, e', \bot) \mid (e, e') \in P \wedge e' \notin \text{Var} \wedge (e, e') \neq (\bot, \bot)\}$.

The user is then prompted for feedback on each triplet in the provenance, and may choose one of the following responses for a given triplet (e, e', v).
`MUST`: this feedback entails that from now on, only formal queries Q for which $(e, e', v) \in \text{prov}(Q', Q, G, \varphi')$ for some binding φ' will be proposed.
`MUST NOT`: only formal queries Q for which $(e, e', v) \notin \text{prov}(Q, Q'', G, \varphi')$ for all bindings φ' will be proposed.
`MAYBE`: no restrictions with respect to the triplet.

Example 5.2. *Reconsider Figure 2; we have already explained (Example 4.2) how the formal query is obtained and how the mapping of elements is based on provenance information. We now also include, in the rightmost column, variable bindings for a query output example, e.g., the binding of* `?s2` *to* `Hyderabad`*. This allows procuring feedback for each triplet, through the checkboxes to their left. Due to our fine-grained provenance tracking, the feedback is highly informative: for instance, the user may confirm that the* `<livesin>` *predicate captures their intention expressed by* `lives_in`*. This will lead to considering only queries which include the triple* (`lives_in`, `<livesin>`, $\bot$)*. In contrast, they may e.g. convey that the assignment of* `Rajnesh_Domalpalli` *to* `?x` *is incorrect, leading to pruning any candidate that includes* (`?x`, `?x`, `Rajnesh_Domalpalli`) *(using our edit operations, every variable is always mapped to itself or* $\bot$*). Such fine-grained feedback, both positive and negative, significantly narrows the search space of possible queries.*

6 Experiments

We have implemented our solution in a prototype called SPARQLIt. The prototype is implemented in .Net, using Blazor [9] for its front-end, Elasticsearch [18] for the Search Engine (used for groundings) and Apache Jena [5] for the Triple Store. All experiments were run on Intel i7-core processors with 32GB of RAM.

PREPRINT

As Knowledge Bases, we have used YAGO [34] English facts (approx. 160M triples) and DBPedia [12]. Since the notion of semi-formal queries is novel, to our knowledge, no existing benchmarks are available. To this end, we have constructed benchmarks based on the first 50 NL questions in the training set of QALD-9 [35]. Their translation to gold formal queries w.r.t. DBpedia is given in [35], and we have formulated the gold queries w.r.t. YAGO; we have stripped aggregation to obtain selection queries, see [11]. We have then constructed two benchmarks of semi-formal queries: (1) *QALD Translated* is based on manual translation that is oblivious to the terminology used in the KBs, e.g., "Who is the tallest player of the Atlanta Falcons?" is translated to the semi-formal selection query `?x type AtlantaFalcons. ?x height ?y` (see [11]); (2) *QALD Cross-KB* uses the formal query for YAGO as a semi-formal query to be evaluated with respect to DBPedia, and vice versa. Intuitively, *QALD Translated* and *QALD Cross-KB* are used to simulate users who are unfamiliar with neither KBs and users who are familiar with one KB and wish to use the other, respectively (user feedback is simulated using the underlying, hidden, formal query). Finally, we have generated a synthetic benchmark, where we have varied different aspects of the query structure, to examine their effect on our solutions.

As solution baselines we have used the NL-to-SPARQL query engine *gAnswer* [23], which achieved the best results in the QALD-9 Challenge [35], as well as multiple variants of our solution: (1) *With-Empty.* A variant which does not prune queries with empty results; (2) *No-Syn.* This variant does not use synonyms for distance computation, unlike our standard implementation (see Section 4); (3) *Top-50.* By default, we configure the Search Engine to return the top-100 results, whereas in this variant it is configured to return only 50.

Our evaluation metrics are the *number of user interactions*, i.e., the number of executions of step 4 in Figure 1; and the *total response time*, i.e., the total computation time of proposed queries, throughout the interactive session. If the query was not found after 50 interactions we consider it a failure.

We next summarize our experimental results.

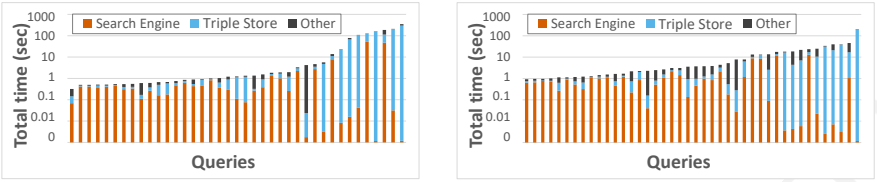
Overall Performance. In Figures 3a and 3b we show the cumulative percentage of formal queries successfully found by the different solutions, for the *QALD Translated* workload and DBpedia, with different bounds on the number of user interactions and total computation time. The results show that SPARQLIt is the most successful solution. In particular, interacting *once* with the user is already sufficient to outperform gAnswer, and to successfully find 52% of the queries; with up to 3 interactions this percentage increases to 62%; and with up to 10 interactions (and up to 23 seconds total time) this percentage increases to 68%. All restricted variants perform worse than SPARQLIt in terms of the number of interactions, showing the effect of our design choices. Figures 3c and 3d show results for the same experiment over the YAGO KB. We exclude gAnswer here since it is tailored for DBpedia. SPARQLIt achieves the best results in both metrics; With-Empty is significantly worse, indicating that the design choice of discarding queries that yield empty results is effective. We summarize the success rates of SPARQLIt using up to 50 interactions in Table 1 (first column), and contrast them with With-Empty and gAnswer.

We have executed the above experiments using the QALD Cross-KB workload (where YAGO queries are used as semi-formal queries over DBpedia and

PREPRINT

	SPARQLIt+ QALD Translated	SPARQLIt+ QALD Cross-KB	With-Empty+ QALD Translated	gAnswer+ QALD-9
DBpedia	78%	81%	64%	36%
YAGO	84%	81%	64%	-

Table 1: Percent of successfully found QALD queries



(a) DBPedia (b) YAGO

Figure 4: Time segmentation for SPARQLIt

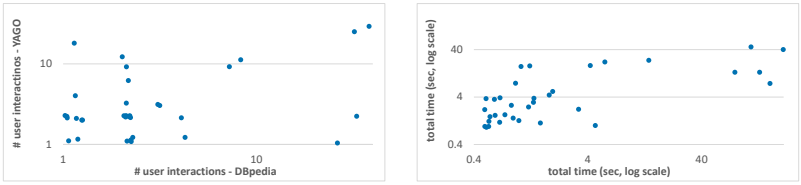
vice versa). The trends were similar: with one interaction, we have successfully found 44% of the queries in both KBs; using up to 3 interactions we found 53% (resp., 56%) of the queries in DBpedia (resp., YAGO); and using up to 10 interactions we found 67% (resp., 65%) of the DBpedia (resp., YAGO) queries. We summarize the success rates with up to 50 interactions in Table 1 (second column). *Overall, SPARQLIt had a high success rate and has succeeded in finding formal queries for the same semi-formal input over different KBs, as well as in finding the same formal query starting from different semi-formal queries.*

Component Breakdown. Figure 4 shows a breakdown of the total computation time to (1) the Search Engine; (2) Triple Store and (3) all other components. The Triple Store and Search Engine are indeed responsible for a large fraction of the overall execution time (median 93% and 86% of the total time, respectively for YAGO and DBpedia); among the two, the time incurred by the Search Engine is typically higher: many query candidates are typically pruned and do do not reach the Triple Store Manager. In contrast, when the overall response time is slower, we observe that it is mainly due to high latency Triple Store queries.

Effect of the KB. Figures 5a and 5b examine the effect of the KB on the difficulty of finding the target query. They show for each query the needed number of interactions (resp., total response time) for DBpedia (x-axis) vs. YAGO (y-axis). The graphs show relatively weak correlation (Pearson correlation coefficient is$\sim$0.6 for both graphs), given that DBpedia and YAGO have many common information sources (most notably, Wikipedia). This serves as evidence that the specifics of the KB structure are indeed essential when writing formal queries.

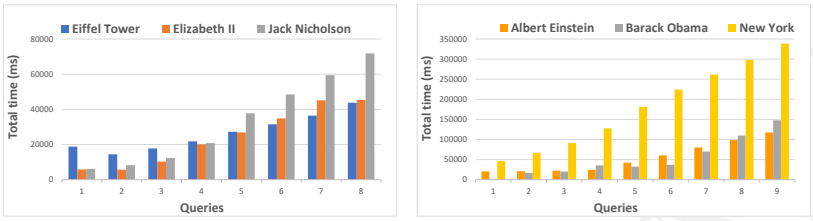
Synthetic Queries. Figure 6 shows the response time of SPARQLIt for representative synthetic queries. Queries include 8 triple patterns structured as chains (subsequent triple patterns share a single variable) or stars (all triple patterns share a single variable), and we vary the number of edits needed to obtain the correct formal query. The response time grows roughly linearly, although the space of relevant structures grows exponentially with the number of edits. This demonstrates the effectiveness of our approach in pruning irrelevant sub-queries.

PREPRINT



(a) # user interactions (b) Total response time

Figure 5: KBs correlation (each point denotes a QALD query)



(a) "Chain"-shaped queries (b) "Star"-shaped queries

Figure 6: Response time vs. number of Subject-Object Switches.

7 Related Work

The SPARQLIt system prototype was demonstrated in ICDE '21 [3], but the short paper accompanying the demonstration does not include details of our solution. Many other lines of research study solutions that assist users in query formulation. In particular, there is a large body of work on NL interfaces over KBs (e.g., [14, 16, 23, 37, 41]) or databases (e.g., [8, 24, 27, 28, 32]). Compared to the NL approach, our solution requires users to provide a more structured specification, yet leverages this structure for an improved interactive process (see Section 6). Another approach is that of Keyword Search over a KG (e.g., [26, 21]); here again a major challenge is recovering from a situation where no suitable query was found, i.e., how to make the process interactive. Other works focus on autocompletion of SPARQL queries (e.g., [17, 30, 40]); these lines of work are complementary to ours: in future work we will study adaptations of auto-completion tools to semi-formal queries, towards incorporating them in our framework. Another relevant line of work focuses on similarity search, studying means of finding, for a given initial query, similar queries that return additional results (e.g., [19, 29, 31, 38, 42]. In particular, Zheng et al. [42] studied semantic similarity search for SPARQL, and introduced an edit distance notion for RDF graphs. While there is some resemblance to our work in the use of structural edit operations, the operations that we consider are different, since we do not require a formal query as input. Instead, we use measures based on syntactic similarity and string similarity. In Query-by-example, queries are reverse-engineered based on positive/negative result examples provided by the user (e.g., [1, 2, 10, 6, 13, 15, 25, 39]). This method can be effective when the users search typed instances for which they can easily provide positive and negative examples, but is typically challenging to use when the query includes non-categorical predicates, which are typically very sparse and heterogeneous, and when users cannot provide sufficient examples. Finally, Faceted (navigational) Search enables users to refine their search options by navigating (drilling) down, and has been studied in the context of RDF querying (e.g., [7, 20, 22, 36]). A challenge for interaction in this context arises when the browsed query parts

PREPRINT

may not match the other intended parts of the user query. If the user performs a sequence of drilling-down steps leading to a "dead-end", it is unclear which steps should be modified and how.

8 Conclusion

We have introduced a novel framework that assists users in querying RDF KBs. Users write queries that do not necessarily match in contents and structure to the KB, and are given proposals for queries that do. Leveraging provenance, the framework procures fine-grained feedback on the proposed queries, guiding the translation. In future research we will extend our Structural Edits operators as well as the fragment of SPARQL we have focused on, including e.g., group-by, aggregation and difference.

Acknowledgements. This work was partly funded by the Israel Science Foundation (grant No. 2015/21) and by the Israel Ministry of Science and Technology.

This preprint has not undergone peer review or any post-submission improvements or corrections. The Version of Record of this contribution is published in [4], and is available online at `https://doi.org/10.1007/978-3-031-12423-5_9`, `https://link.springer.com/chapter/10.1007/978-3-031-12423-5_9`.

References

[1] A. Abouzied, D. Angluin, C. H. Papadimitriou, J. M. Hellerstein, and A. Silberschatz. Learning and verifying quantified Boolean queries by example. In *PODS*, 2013.

[2] E. Abramovitz, D. Deutch, and A. Gilad. Interactive inference of SPARQL queries using provenance. In *ICDE*, 2018.

[3] Y. Amsterdamer and Y. Callen. Sparqlit: Interactive SPARQL query refinement. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 2649–2652. IEEE, 2021.

[4] Y. Amsterdamer and Y. Callen. Provenance-based SPARQL query formulation. In *DEXA*, volume 13426, 2022.

[5] Apache Jena. `https://jena.apache.org/`.

[6] M. Arenas, G. I. Diaz, and E. V. Kostylev. Reverse engineering SPARQL queries. In *WWW*, 2016.

[7] M. Arenas, B. C. Grau, E. Kharlamov, S. Marciuska, D. Zheleznyakov, and E. Jiménez-Ruiz. SemFacet: semantic faceted search over YAGO. In *WWW*, 2014.

[8] C. Baik, H. V. Jagadish, and Y. Li. Bridging the semantic gap with SQL query logs in natural language interfaces to databases. In *ICDE*, pages 374–385. IEEE, 2019.

PREPRINT

[9] Blazor. https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor/.

[10] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive inference of join queries. In *EDBT*, pages 451–462, 2014.

[11] Code and query repository for SPARQLIt. https://github.com/ycallen/dexa22.

[12] DBpedia. https://wiki.dbpedia.org/.

[13] G. I. Diaz, M. Arenas, and M. Benedikt. SPARQLByE: Querying RDF data by example. *PVLDB*, 9(13), 2016.

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

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

[16] M. Dubey, S. Dasgupta, A. Sharma, K. Höffner, and J. Lehmann. Asknow: A framework for natural language query formalization in SPARQL. In H. Sack, E. Blomqvist, M. d'Aquin, C. Ghidini, S. P. Ponzetto, and C. Lange, editors, *ESWC*, 2016.

[17] A. El-Roby, K. Ammar, A. Aboulnaga, and J. Lin. Sapphire: Querying RDF data made simple. *PVLDB*, 9(13), 2016.

[18] Elasticsearch. https://www.elastic.co/elasticsearch/.

[19] S. Elbassuoni, M. Ramanath, and G. Weikum. Query relaxation for entity-relationship search. In *ESWC*, 2011.

[20] S. Ferré. Expressive and scalable query-based faceted search over SPARQL endpoints. In *ISWC*, volume 8797, 2014.

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

[22] F. Haag, S. Lohmann, S. Siek, and T. Ertl. Queryvowl: Visual composition of SPARQL queries. In *ESWC*, volume 9341, 2015.

[23] S. Hu, L. Zou, J. X. Yu, H. Wang, and D. Zhao. Answering natural language questions by subgraph matching over knowledge graphs. *IEEE Trans. Knowl. Data Eng.*, 30(5), 2018.

[24] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. Learning a neural semantic parser from user feedback. In R. Barzilay and M. Kan, editors, *ACL*, 2017.

[25] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri. Querying knowledge graphs by example entity tuples. In *ICDE*, 2016.

PREPRINT

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

[27] H. Kim, B. So, W. Han, and H. Lee. Natural language to SQL: where are we today? *PVLDB*, 13(10), 2020.

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

[29] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: Give me an example of what you need. *PVLDB*, 7(5), 2014.

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

[31] R. Schenkel, A. Theobald, and G. Weikum. Semantic similarity search on semistructured data with the XXL search engine. *Inf. Retr.*, 8(4), 2005.

[32] J. Sen, C. Lei, A. Quamar, F. Özcan, V. Efthymiou, A. Dalmia, G. Stager, A. R. Mittal, D. Saha, and K. Sankaranarayanan. ATHENA++: natural language querying for complex nested SQL queries. *PVLDB*, 13(11), 2020.

[33] SPARQL query language for RDF. `https://www.w3.org/TR/rdf-sparql-query/`.

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

[35] R. Usbeck, R. H. Gusmita, A. N. Ngomo, and M. Saleem. 9th challenge on question answering over linked data (QALD-9). In *ISWC*, volume 2241 of *CEUR Workshop Proceedings*, 2018.

[36] H. Vargas, C. B. Aranda, A. Hogan, and C. López. RDF explorer: A visual SPARQL query builder. In *ISWC*, volume 11778, 2019.

[37] D. Vollmers, R. Jalota, D. Moussallem, H. Topiwala, A. N. Ngomo, and R. Usbeck. Knowledge graph question answering using graph-pattern isomorphism. *CoRR*, abs/2103.06752, 2021.

[38] Y. Wang, A. Khan, T. Wu, J. Jin, and H. Yan. Semantic guided and response times bounded top-k similarity search over knowledge graphs. In *ICDE*, 2020.

[39] Y. Y. Weiss and S. Cohen. Reverse engineering SPJ-queries from examples. In E. Sallinger, J. V. den Bussche, and F. Geerts, editors, *PODS*, 2017.

[40] Wikidata. `https://www.wikidata.org/wiki`.

[41] X. Yin, D. Gromann, and S. Rudolph. Neural machine translating from natural language to SPARQL. *Future Gener. Comput. Syst.*, 117, 2021.

[42] W. Zheng, L. Zou, W. Peng, X. Yan, S. Song, and D. Zhao. Semantic SPARQL similarity search over RDF knowledge graphs. *PVLDB*, 9(11), 2016.

PREPRINT