Interactive SPARQL Query Formulation using Provenance

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 graph and require only basic familiarity with SPARQL. This means that the user-intended query over the knowledge graph 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 knowledge graph. 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 graphs (KGs)[1] such as YAGO [49], DBpedia [32] and others. Such KGs can be queried using SPARQL, the W3C standard query language,[2] which has a relatively simple syntax, and multiple graphical interfaces that have been developed (e.g., [22, 41, 31, 45, 53]) to further simplify query specification. Familiarity with the SPARQL syntax, however, is not sufficient for using it: writing a SPARQL query requires a deep understanding of the KG content and structure. In turn, in many useful KGs, this structure is highly complex and contains many irregularities. The KG structure often further evolves over time, and so even users who are initially familiar with it may struggle to adapt to such modifications.[3]

Many existing tools were developed to address the challenge of SPARQL Query formulation, including natural language interfaces (e.g., [9, 14, 16, 23, 43, 54, 56, 61]), auto-complete suggestions (e.g., [12, 17, 44]), query-correction proposals (e.g., [17]) and query-by-example tools (e.g., [2, 6, 13, 27]). See discussion in Section 8. Many challenges yielded by the schema complexity, however, remain. Specifically, due to the schema complexity, queries that are generated

[1] RDF - Semantic Web Standards. `https://www.w3.org/RDF/`

[2] SPARQL Query Language for RDF. `https://www.w3.org/TR/rdf-sparql-query/`

[3] For convenience, in this extended version, added material is highlighted in blue, and revised material is highlighted in orange.

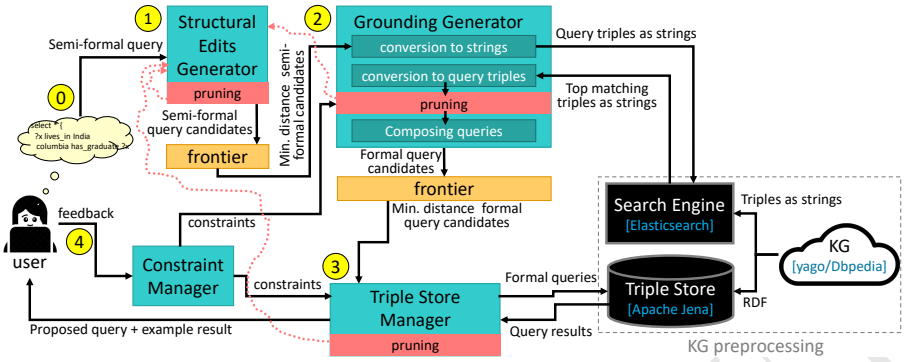


Figure 1: Overall architecture of SPARQLIt

and proposed by such tools may not match the user intention, or not return the expected results when evaluated with respect to the KG. Recovering from such cases is quite difficult, since the users may not even know what went wrong.

Solution overview. To this end, we propose a novel solution that assists non-experts in querying KGs. Our architecture is depicted in Figure 1. We describe it here at a high level, and elaborate on each component in the following paragraphs. The user starts (step (0), on the left) by writing what we call a "semi-formal query". This is a query that consists of triples, like in SPARQL, but whose contents – entities and properties – do not necessarily match any entity/property name in the KG. 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. One reason is that Columbia is represented in YAGO as `<Columbia_University>` in YAGO. The other, more elusive reason, is that `livesIn` is usually populated with cities rather than countries, which is difficult for a user to discover even using query formulation tools. To this end, our solution generates candidate formal queries, i.e., queries that match the KG. 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 livesIn ?y. ?y isLocatedIn India`; and *groundings* (step 2) where elements of semi-formal queries are replaced by elements of the KG, e.g., replacing `columbia` by `<Columbia_University>`. Having constructed candidate formal queries, the *Triple Store Manager* queries a Triple Store to find example results for these queries, and to prune queries that include unwanted results or omit required results (step (3)).

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 KG. 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 KG elements). See bottom pane of the screenshots in 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 4) 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 next briefly describe each of the modules, to be described in further details in the sections that follow.

Structural Edits Generator (Section 3). The structure of the input semi-formal query may not match the underlying KG: the subject-object order in predicates may be reversed; a semantic relationship captured by a single triple in the semi-formal query may be captured by a chain of triples in the KG; etc. We formally define a set of structural edit operations that transform one semi-formal query into another, associate a cost with each edit and then greedily traverse the space of semi-formal queries obtained via sequences of such edits, by increasing order of edit distance. Importantly, we use provenance to keep track of the performed edits, and to map each part of input semi-formal query to the part of the candidate semi-formal query that corresponds to it. Constraints on the provenance are accumulated to avoid re-examination of the same semi-formal query, and prune other semi-formal queries based on their relationship to already examined queries.

Groundings Generator (Section 4). Each candidate semi-formal query may be grounded in different ways, where a grounding corresponds to transforming each non-variable element in the semi-formal query to an element from the KG. We treat elements as strings, and use a string search engine (Elasticsearch[4]) to find these KG elements. A subtlety is that because the search engine is unaware of the query structure, its order of results in string search is only weakly correlated with the expected order in element search. For that, we post-process the search engine results, re-ordering them based on a distance function that is tailored to our setting. We again use provenance to keep track of grounding transformations. We also apply pruning of queries for which the search engine yields no matches, or for which the provenance does not match the constraints derived from past user feedback.

Triple Store Manager (Section 5). Each candidate formal query is fed to the Triple Store Manager, which accesses a triple store such as Jena[5] for query evaluation. We allow pruning, by default, queries whose evaluation yields no results. We combine here the standard provenance for query output with

[4]Elasticsearch. `https://www.elastic.co/elasticsearch/`
[5]Apache Jena. `https://jena.apache.org/`

our above mentioned provenance for query transformations, and then further prune queries that do not match the constraints on the output, i.e., contain rejected results or do not include required results. For efficiency, pruning is performed in batches, grouping together queries with similar structure to check, simultaneously, if they contain relevant results. If the query result is non-empty, examples of outputs are chosen to be presented to the user.

User Feedback and Constraints (Section 6). The tracking of provenance throughout our modules allows us to present a comprehensive explanation of the candidate formal queries, in the following form: for each element of the semi-formal query provided by the user, we present the corresponding element in the formal query; if the element is a variable, we further present its binding in the example query result. See Figure 2 for an example (observe that some elements in the formal query have no counterpart in the semi-formal query and vice versa). This explanation is crucial in allowing users to provide fine-grained feedback at the level of triples, indicating that some parts must or must not appear in the generated formal query. This feedback accumulates and serves to form constraints on the generated queries in the iterations that follow. Constraints are expressed and verified in terms of the transformation provenance.

Experiments (Section 7). We implement SPARQLIt in a prototype system and examine its performance over two large-scale KGs, YAGO [49] and DBpedia [32] and a standard query benchmark [52]. Most importantly, the experimental results support the practicality of our approach, requiring only few interactions and a few seconds overall to find most of the examined queries. This is a significant improvement compared to other solutions where, if the user receives a wrong query proposal, there is no methodical way of interacting with the system to find the right query. We also isolate and demonstrate the benefit of the optimizations that we employ. We then dive in to examine the different factors affecting the performance of query proposal computation. We show that the KG plays a central role here, since there is almost no correlation between the difficulty of queries (many user interactions/ long computation time) in YAGO to the difficulty of the same queries in DBpedia.

We overview related work in Section 8 and conclude in Section 9. This paper is an extended version of our work published in [5], and our use of the prototype system described below was demonstrated in [4].

2 Model

We start with preliminaries on RDF and then introduce our notions of semi-formal queries and provenance. For convenience, the main notations used in the sequel are summarized in Table 1.

2.1 RDF and Queries

RDF Knowledge Graphs. An RDF Knowledge Graph (KG) 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.,

Ent	Domain of entities in formal queries
Pred	Domain of predicates in formal queries
Pred	Domain of literals in formal queries
Var	Domain of query variable names
$Q = (G_Q, V_Q)$	Selection query, where G_Q is a basic graph pattern (set of triples with variables) and V_Q is the set of output variables
Ent^{sf}	Domain of entities in semi-formal queries
Pred^{sf}	Domain of predicates in semi-formal queries
Lit^{sf}	Domain of literals in semi-formal queries
Temp^{sf}	Domain of placeholders in semi-formal queries
φ	Assignment of elements to variables
$\varphi(G_Q)$	Set of triples where variables are replaced by their assignment in φ
$Q(G)$	Result of formal query Q over KG G
$\text{prov}(Q, G, A)$	Query answer provenance, the set of all bindings of Q in G yielding A.
$\text{prov}(Q, Q') = (P, C)$	Query provenance of transforming query Q to query Q', where P is a set of element mappings and C is the total cost

Table 1: Notation overview

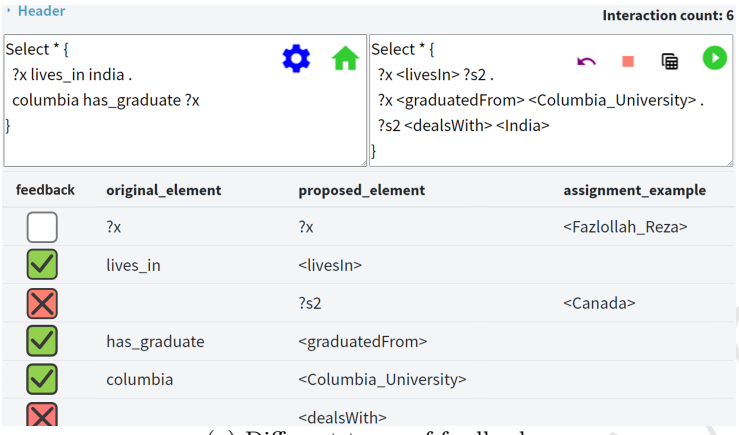
2021). Let Pred be a domain of predicate names (e.g., `livesIn`). An *RDF knowledge base* is a set of triples of the form (s, p, o) where $s \in \text{Ent}$ is the subject, $p \in \text{Pred}$ is the predicate and $o \in \text{Ent} \cup \text{Lit}$ is the object. We use *element* to refer uniformly to 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).

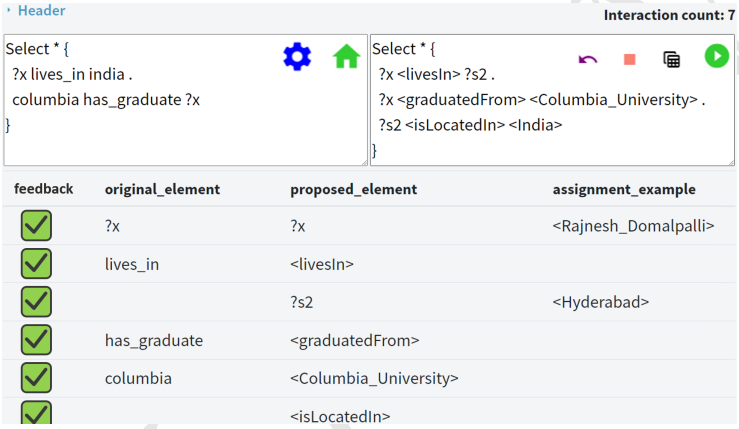
Definition 2.1 (SPARQL selection queries)**.** *Let* $\text{Var} = \{$ `?x`, `?y`, $\dots\}$ *be a set of variables. A* Triple Pattern *is a member of the set* $(\text{Ent} \cup \text{Var}) \times (\text{Pred} \cup \text{Var}) \times (\text{Ent} \cup \text{Lit} \cup \text{Var})$*. Namely, in a triple 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 KGs 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*.*

The general SPARQL syntax is broader, and allows for filtering conditions, difference, and group by – but we focus here on selection queries as the most difficult part for a user unfamiliar with the KG contents. We will assume, in our architecture, a standard SPARQL query engine that also supports more advanced features of RDF such as inference rules. We shall therefore regard, for simplicity, the KG as containing all the facts that can be inferred from it, and comment on inference when relevant.

Query evaluation. Given a formal SPARQL selection query $Q = G_Q, V_Q)$ and an RDF KG G, let φ be a mapping of all variables in G_Q to RDF elements 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 KG 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.



(a) Different types of feedback



(b) User-intended query

Figure 2: SPARQLIt User Interface

2.2 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 placeholder elements from $\text{Temp}^{sf} = \{\texttt{??X}, \texttt{??Y}, \dots\}$ to be replaced by any KG term (entity/literal/predicate) when we transform the query into a formal one (see below). Formally,

Definition 2.2 (Semi-formal queries)**.** *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 2.3. *Figure 2 shows two screenshots of SPARQLIt, in each of which 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 KG (YAGO). The top-right panel displays a candidate formal query, a different candidate each time, that matches the underlying KG, in this case YAGO. (Ignore, for now, the bottom panel.)*

2.3 Provenance Model

We introduce two types of provenance. The first captures "standard" query-to-answer binding.

Definition 2.4 (Query answer provenance)**.** *The provenance of a binding φ obtained by evaluating a formal SPARQL Query Q over a KG 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. In order to prune irrelevant candidate queries effectively, we use fine-grained user feedback at the level of query parts. For that, instead of attaining feedback solely on elements of the candidate query, we ask whether elements of the input query were correctly transformed into elements of the candidate query. This allows us (a) to explain to users why certain query elements appear in the candidate query; (b) to "affix" correct transformations and use them in subsequent query candidates; (c) to avoid incorrect transformations without entirely avoiding an element; and (d) incorporate, in our query cost, the distance between corresponding elements.This is demonstrated in the next example.

Example 2.5. *Consider the running example semi-formal query (top-left of Figure 2a), and the candidate formal query*

```
Select * {?x <graduatedFrom> <Indiana_University>.}
         <Cole_Umbria> <graduatedFrom> ?x.}
```

At first glance, the meaning of this query and its connection to the input query may be unclear. Now, consider the following display of the corresponding elements of the input and output query elements.

Original Element	*Proposed Element*
`lives_in`	`<graduatedFrom>`
`india`	`<Indiana_University>`
`columbia`	`<graduatedFrom>`
`has_graduate`	`<Cole_Umbria>`

The above display hints at what went wrong (e.g., `india` *was wrongly transformed to* `<Indiana_University>` *due to their similarity). At this point, providing feedback at the element level is less meaningful: in particular,* `<graduatedFrom>` *is used twice, once incorrectly and once correctly. Should the users approve or reject this element? In contrast, using feedback at the transformation level allows users to specify "*`lives_in` *is* incorrectly *grounded as* `<graduatedFrom>`*" and "*`has_graduate` *is* correctly *grounded as* `<graduatedFrom>`*". In subsequent candidate queries, we can narrow our search to queries that include the correct transformation and avoid the wrong ones.*

Source	Can be mapped to
$e \in \text{Ent} \cup \text{Pred} \cup \text{Lit}$	e (self), $\bot$
$e \in \text{Var}$	e (self), $\bot$
$e \in \text{Temp}^{sf}$	e (self), $\text{Ent} \cup \text{Pred} \cup \text{Lit}$, $\bot$
$e \in \text{Ent}^{sf} \cup \text{Pred}^{sf} \cup \text{Lit}^{sf} - \text{Ent} \cup \text{Pred} \cup \text{Lit}$	e (self), $\text{Ent} \cup \text{Pred} \cup \text{Lit}$, $\bot$
$e = \bot$	$\text{Ent}, \text{Pred}, \text{Lit}$, fresh Var, fresh Temp^{sf} , $\bot$

Table 2: Allowed sound transformations by element type. A "fresh" Var/Temp^{sf} is a variable/placeholder that does not occur in the source query.

We next define the second type of provenance, where element pairs encode "mappings" of individual elements in Q to the corresponding elements in Q'.

Definition 2.6 (Query transformation provenance)**.** *Let $Q = (G_Q, V_Q)$ and $Q' = (G_{Q'}, V_{Q'})$ be two (formal or semi-formal) queries. A* provenance expression *for a transformation of Q to Q' is denoted by* $\text{prov}(Q, Q') = (P, C)$*: P, referred to as the* transformation mappings*, is a set of pairs (e, e') where either $e = \bot$ or $e \in T \in G_Q$ is an element of a triple T of the BGP of Q, and similarly either $e' = \bot$ or $e \in T \in G_{Q'}$. $C \in \mathtt{N}$ is the* transformation cost*.*

The notation $\bot$ is used to mark deletions/insertions of elements. This means that the used elements as well as the topology of candidate queries may differ from the input query. We discuss the computation of cost values in Sections 3 and 4.

We next define two sets of requirements from the mappings of queries. The first set restricts the mapping to be bijective, i.e., every query element is mapped to exactly one element of the other query (or added/deleted). Intuitively, these requirements ensure that the transformation of every query element is explained, and that feedback can be given at the level of single elements rather than sets. Formally, given $Q = (G_Q, V_Q)$ and $Q' = (G_{Q'}, V_{Q'})$ with $\text{prov}(Q, Q') = (P, C)$, we say that P is *valid* if it satisfies the following.

- *Functionality.* For every triple $T \in G_Q$, for every element $e \in T$, there exists *exactly one* pair $(e, e') \in P$, for some e'.
- *Inverse functionality.* For every triple $T' \in G_{Q'}$, for every element $e' \in T'$, there exists *exactly one* pair $(e, e') \in P$, for some e.

The second set of requirements relates to the gradual process of transforming an informal query to a formal one, ensuring, e.g., that formal elements are not transformed into non-formal ones. We say that P is *sound* if every $(e, e') \in P$ matches the possible mappings listed in Table 2.

Intuitively, every type of query element can be deleted by a transformation or left unchanged.

In our search of formal queries, we will track changes to queries via mapping compositions, which we can show to preserve validity.

Definition 2.7 (Mapping composition)**.** *Given three (formal or semi-formal queries) Q_0, Q_1, Q_2, $\text{prov}(Q_0, Q_1) = (P_1, C_1)$ and $\text{prov}(Q_1, Q_2) = (P_2, C_2)$, the mapping composition $P_2 \circ P_1$ is a set of element pairs such that*

1. *For every* $e_1 \neq \bot$ *such that* $(e_0, e_1) \in P_1$ *and* $(e_1, e_2) \in P_2$*, it holds that* $(e_0, e_2) \in P_2 \circ P_1$
2. *For every* $(e_0, \bot) \in P_1$*, it holds that* $(e_0, \bot) \in P_2 \circ P_1$
3. *For every* $(\bot, e_2) \in P_2$*, it holds that* $(\bot, e_2) \in P_2 \circ P_1$
4. *For every* $(e_0, e_2) \in P_2 \circ P_1$*, it either holds that* $(e_0, e_1) \in P_1$ *and* $(e_1, e_2) \in P_2$ *for some* $e_1 \neq \bot$*, or that* $(e_0, e_2) \in P_1$ *and* $e_2 = \bot$*, or that* $(e_0, e_2) \in P_2$ *and* $e_0 = \bot$*.*

The composition provenance is then $\text{prov}(Q_0, Q_2) = (P_2 \circ P_1, C_1 + C_2)$*.*

We can now show:

Proposition 2.8. *Let* Q_0, Q_1, Q_2 *be three (formal or semi-formal queries), such that* $\text{prov}(Q_0, Q_1) = (P_1, C_1)$ *and* $\text{prov}(Q_1, Q_2) = (P_2, C_2)$*. The mapping composition* $P_2 \circ P_1$ *is a mapping from* Q_0 *to* Q_2*; if* P_1 *and* P_2 *are valid then* $P_2 \circ P_1$ *is valid; and if* P_1 *and* P_2 *are sound then* $P_2 \circ P_1$ *is sound up to variable freshness.*

Proof. $\Rightarrow$ *$P_2 \circ P_1$ is a mapping.* $P_2 \circ P_1$ is a set of pairs (e_0, e_2) such that, by Def. 2.7 item 4, e_0 is either $\bot$ or $(e_0, e1)$ in P_1, and then by definition e_0 is an element of Q_0 (or $\bot$). By a symmetric argument, e_2 is either $\bot$ or an element of Q_2.

$\Rightarrow$ *Composition preserves validity.* Assume that P_1 and P_2 are valid. We need to show functionality and inverse functionality for $P_2 \circ P_1$. Let e_0 be an element of Q_0. By the functionality of P_1 there is exactly one pair $(e_0, e_1) \in P_1$. If $e_1 = \bot$, by item 2 of Def. 2.7, $(e_0, \bot) \in P_2 \circ P_1$; and otherwise, e_1 is an element of Q_1 and then by the functionality of P_2, there is exactly one pair (e_1, e_2) in P_2, and by item 1 of Def. 2.7 we have that $(e_0, e_2) \in P_2 \circ P_1$. This shows there is *at least* one (e_0, e_2) for each element e_0 of Q_0. We now need to show it is *unique*. Assume by contradiction that for some $e_0 \neq \bot$ and some $e_2 \neq e_2'$ we have $(e_0, e_2), (e_0, e_2') \in P_2 \circ P_1$. W.l.o.g assume that $e_2 \neq \bot$. A pair of non-$\bot$ elements may only occur in $P_2 \circ P_1$, according to item 4 above, if $(e_0, e_1) \in P_1$ and $(e_1, e_2) \in P_2$ for some $e_1 \neq \bot$. Then, by the functionality of P_1 and inverse functionality of P_2, respectively, there is at most one e_1 such that $(e_0, e_1) \in P_1$ and such that $(e_1, e_2) \in P_2$. Therefore, e_2' cannot be an element of Q_2, so it must be $\bot$, and, by item 4, $(e_0, \bot) \in P_1$. This, together with $(e_0, e_1) \in P_1$, contradicts the functionality of P_1. This proves (by contradiction) that $P_2 \circ P_1$ is functional. Inverse functionality is proved by a symmetric argument. Consequently, $P_2 \circ P_1$ is indeed valid.

$\Rightarrow$ *Composition preserves soundness.* Assume that P_1 and P_2 are sound. We will analyze the soundness of $P_2 \circ P_1$ by case for any pair $(e_0, e_2) \in P_2 \circ P_1$. If $e_0 = \bot$, e_2 can be of any type or value (we do not prove freshness in case of a variable or placeholder; see discussion below). If $e_2 = \bot$, e_0 can be of any type without contradicting soundness. Otherwise, $e_0, e_2 \neq \bot$ and for some $e_1 \neq \bot$, we have that $(e_0, e_1) \in P_1$, $(e_1, e_2) \in P_2$. We can now ignore cases when $e_0 = e_1$ or $e_1 = e_2$, because then (e_0, e_2) is either in P_1 or P_2 and does not contradict soundness. We are left with the case when $e_0\, e_2 \neq e_1$ and $e_0, e_1, e_2 \neq \bot$, but this is impossible by Table 2: if we start with a e_0 that is formal element or variable, it can only be deleted ($e_1 = \bot$); and if e_0 is a non-formal or placeholder element, it can only be transformed to a formal element $e_1 \in \text{Ent} \cup \text{Pred} \cup \text{Lit}$, but then e_1 can only be deleted ($e_2 = \bot$). Hence, $P_2 \circ P_1$ is sound. □

Note that mapping composition does not necessarily follow the requirement in Table 2 that added variables/placeholders are fresh: if for some variable/placeholder e_2, $(e_2, e_1) \in P_1$ and $(\bot, e_2) \in P_2$, then $(e_2, e_1), (\bot, e_2) \in P_2 \circ P_1$. e_2 is fresh with respect to Q_1 but not with respect to Q_0. This issue can be technically solved by ensuring that newly added variables/placeholders have unique names.

We also note that our mappings are oblivious to the number of occurrences of an element in a query and to the positions of these occurrences within the query. As we show below (in Section 6), this enables a simple yet expressive and fine-grained form of user feedback.

In the following sections, we introduce different transformations on query structures and contents. The above defined provenance will serve to track sequences of transformations, the connections between the components of the resulting query to the input query, and to compute the transformation cost.

3 Structural Edits

A first type of edits that we apply to semi-formal queries is geared towards modifying the query structure. Edits are applied to triple patterns, capturing, intuitively reordering/deletion/insertion of query elements. 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 2.6), but keep the costs abstract at this point and discuss concrete cost choices below.

Subject-Object Switching: switch the subject and object of a query triple $T = \langle s, p, o \rangle$, yielding a new triple $T' = \langle o, p, s \rangle$. 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.

Element Exclusion: replace a formal/semi-formal subject/object/predicate of a query triple $T = \langle s, p, o \rangle$ 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 KG. 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 intuitively means that the candidate query will not include an element corresponding to 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 predicate by a path with two predicates. The provenance includes (s, s), (p, p), $(\bot, \texttt{?x})$, $(\bot, \texttt{??Y})$, (o, o), and (e, e) for every other element e.

We have chosen the three operations above to support common cases of reordering, insertion and exclusion required to match a semi-formal query to a formal one. In our experimental study (see Section 7), we show that the combination of these operations is already sufficient to support the translation of most of the informal queries we have tested. We discuss additional operations below.

Example 3.1 (Structural edit operations)**.** *Consider the triple pattern* `columbia has_graduate ?x`*. If an inverse predicate is used in the KG, a candidate query may be generated by Subject-Object Switching yielding* `?x has_graduate columbia`*. This will allow the triple to be later grounded, e.g., to* `?x graduatedFrom Columbia_University`*.*

Applying Element Exclusion to the same triple could yield `columbia ?p ?x` *or* `?y has_graduate ?x`*, generalizing the query by placing a variable that may be bound to any predicate in the former case, and to any entity in the latter case. For example, if the KG does not include any predicate corresponding to "has graduate", we may relax the requirement to retrieving Indian residents that are related in any way to Columbia University.*

Last, the connection between institutes and graduates in the KG may be expressed by a 2-edge path rather than one, e.g., using the thesis as an intermediate node. Applying Predicate Splitting on `columbia has_graduate ?x` *would yield the two triple patterns* `columbia has_graduate ?s1. ?s1 ??P1 ?x`*. These triples may then be later grounded, e.g., to* `Columbia_University hasGraduateThesis ?s1. ?s1 author ?x`*, in particular by replacing the placeholder* `??P1` *by a concrete predicate name.*

By Prop. 2.8, if a single edit operation produces a mapping that fulfills the validity and soundness requirements, then a sequence of such edit operations will also produce a composed mapping that is valid and sound. We can show that each of the three operations above indeed produces a valid and sound mapping.

Proposition 3.2. *The operations Subject-Object Switching, Element Exclusion and Predicate Splitting, as defined above, produce valid and sound transformation mapping.*

Proof. Let $\text{prov}(Q, Q') = (P, C)$ be the provenance resulting from a single structural edit operation on a query Q. We now show that P is valid and sound according to the requirements in Section 2.3, for each of the three structural edit operators defined above. Per element e, if e occurs only in the identity mapping (e, e) in P, then e does not violate validity and soundness. As Subject-Object Switching produces the identity mapping, we can conclude that it produces valid and sound transformation mapping. Next, we need to focus only on pairs (e, e') with $e \neq e'$ in the other operations. Element Exclusion produces the pairs $(s, \bot)$, $(\bot, \texttt{?x})$, i.e., each element in the input and/output query is mapped to exactly one element (validity). Since `?x` is fresh, both mappings are sound by Table 2 (soundness). Finally, Predicate Splitting produces the pairs $(\bot, \texttt{?x})$, $(\bot, \texttt{??Y})$ – the insertion of a fresh variable and a fresh placeholder – which follow the requirements of both validity and soundness. □

Example 3.3 (Sequences of edit operations)**.** *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 the identity mappings for the rest of the query elements.*

```
NextSemiFormalQuery (Frontier, Cahce, constraintsMgr)
1   iterator ← Frontier . iterator();
2   Q_candidate ← iterator . next();
3   while Q_candidate! = null and !Q_candidate. isActive() do
4       if !Q_candidate. hasNextOperator() then
            Frontier . remove(Q_curr);
5           Q_candidate ← iterator . next();
6       else
7           Q_new ← Q_curr. applyNextOperator();
8           Q_next ← iterator . peek();
9           if ! Cache . contains(Q_new) then
10              Cache . add(Q_new);
11              Frontier . add(Q_new);
12              if constraintsMgr . isPruned(Q_new) then
13                  Q_new. setInactive();
14              else if Q_next = null or Q_new. score < Q_next. score then
15                  return Q_new;
16          if Q_next! = null and Q_new. score > Q_next. score then
17              Q_candidate ← iterator . next();
18  return Q_candidate;
```

Algorithm 1: Getting the next semi-formal candidate.

Searching for structural edits. We store a *frontier* of semi-formal queries, ordered by increasing cost and initially including only the input semi-formal query. Whenever prompted, the Generator operates similarly to Dijkstra's minimum-distance path search algorithm, to find the next least costly semi-formal structure, as described in Algorithm 1.

We next overview the steps of Algorithm 1. We start iterating over the Frontier (lines 1-2). If the current candidate semi-formal query, $Q_{\text{candidate}}$, is not active, i.e., it was already pruned, we check if there is any additional operator we can apply to it to get a new candidate query. If not (line 4), we remove it from the Frontier. Otherwise, we try to generate the next minimal-cost candidate query, by applying the next minimal-cost operator that was not applied yet to $Q_{\text{candidate}}$; for example, we can try to apply Subject-Object Switching to each triple of the query, then Predicate splitting to each triple, etc. This yields a semi-formal query Q_{new} (line 7). We then check if we have already seen and cached Q_{new} (line 9): this can occur since the same candidate semi-formal query can be computed via different operator sequences. E.g., we can compute the same query by applying Subject-Object Switching once or three times on the same triple. Next (line 12), we check if the query is pruned by the constraints in the Constraints Manager. If it is, we set Q_{new} to be inactive, which means it will not be returned as a semi-formal query candidate, but we may use it to construct other semi-formal queries in the following iterations of the algorithm. If Q_{new} is not pruned, and there is no following query in the Frontier with lower cost, then we return Q_{new} as the minimal cost query candidate (lines 14-15). Indeed, the next query on the Frontier is the minimal cost of all the queries in the Frontier, and queries generated by applying further operations only have larger costs. Otherwise (line 16), if there is a following query in the Frontier with a lower score, we repeat the process, until we find a query candidate, or

we have no more candidates left in the Frontier ($Q_{\text{candidate}} = \text{null}$).

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 empirically obtained superior results with costs $2C$, $100C$ and $30C$ for Subject-Object Switching, Element Exclusion and Predicate Splitting, respectively. The cost of a sequence of operations is defined to be the sum of individual operation costs. A candidate query can be obtained via different sequences of operations with different total costs. However, by the Dijkstra-like algorithm we employ, the first time a query is returned by the algorithm is when it has a minimal weight. Then, by caching the considered queries, we can avoid reconsidering it in the future.

4 Grounding Generator

The Grounding Generator gets as input a semi-formal query Q' and the KG 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 an off-the-shelf *search engine* (e.g., Elasticsearch) that indexes triples from the KG. The engine returns the top-k relevant KG 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 KG 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 KG triples* `<Aadya> <livesIn> <India>`, `<Aarav> <livesIn> <Indianapolis>`. *Since the original triple pattern has* `?x` *as subject, we replace the subject in the KG 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.

Provenance revisited. Recall that only triple patterns with consistent provenance expressions were combined. The overall provenance is then defined as follows: its pair set is the union of pair 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 Example 3.3, we generate candidate groundings for* `lives_In`*,* `India`*,* `has_graduate`*,* `columbia` *and the placeholder* `??P1`*. A formal query resulting from one such combination of groundings is shown on the top-right part of Figure 2 with its provenance on the bottom: e.g.,* `has_graduate` *has transformed to* `<graduatedFrom>`*, while* `isLocatedIn` *is newly added (so it has no 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. To account for *semantic synonyms* that are represented by very different strings, we generate a set of synonyms (using `https://www.datamuse.com/api/`) for each string, and take the minimal edit distance between a synonym of $s(t)$ and a synonym of $s(t')$. We revisit this design choice in Section 7. 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 KG, multiplied by the number of query terms.

5 Triple Store Manager

The Triple Store Manager receives as input a formal SPARQL query produced by the Grounding Generator, and executes it over a triple store to obtain query results. If the query result is non-empty, it chooses an example result with 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 KG, $A \in Q(G)$ an answer, φ a binding yielding A and* $\text{prov}(Q, G, \varphi)$ *its provenance. The provenance* $\text{prov}(Q', Q, G, \varphi) = (P', C)$ *where* $P' = \{(e, e', v) \mid (e, e') \in P \wedge e' \in \text{Var} \wedge (e', v) \in \text{prov}(Q, G, \varphi)\} \cup \{(e, e', \bot) \mid (e, e') \in P \wedge e' \notin \text{Var} \wedge (e, e') \neq (\bot, \bot)\}$.

Query Pruning and Optimizations. The user may choose to receive only proposals for queries whose result is non-empty, and may provide feedback on required/prohibited query results (see Section 6). In this case, the Triple Store Manager serves as a key component in pruning candidate queries: we observed that even with the pruning performed by previous modules, typically only a small fraction of the queries that get to this stage have actual results in the KG. The following main optimizations are employed to this end: first, instead of individually executing each grounding, we group together queries with the same structure and generate a query where each element is replaced by a special variable restricted be one of the candidate groundings of this position. The results then include the results of all the combined candidate queries (and more) and we can find for each result the query that yields it by considering the values assigned to the added special variables (see example below). A downside that we observe is that execution of the resulting queries is often not well optimized by the query engine. To this end, we rely on the monotonicity of SPARQL selection queries, and start by executing only two triples of the query.[6] Only if this query has a result, we remove the irrelevant groundings, and add a third triple, and so on. This method of gradual execution gives an additional major benefit of caching sub-queries with empty results, and pruning subsequent queries that include these sub-queries at the Grounding Generator stage.

Example 5.2. *Assume that for the semi-formal query from Example 2.3, we have obtained two candidate formal queries: one has the BGP* `?x livesIn <India>. <Columbia_University> <hasWebsite> ?x.`*, and the other has the BGP* `?x livesIn <Indianapolis>. <Cole_Umbria> <graduatedFrom> ?x.` *The Triple Store Manager can batch together these two queries, since they have the same structure. The executed query would then be*

```
Select * {
  ?x livesIn ?o1.
  FILTER(?o1 = <India> || ?o1 = <Indianapolis>).
  ?s1 ?p1 ?x.
  FILTER(?s1 = <Cole_Umbria> || ?s1 = <Columbia_University>).
  FILTER(?p1 = <graduatedFrom> || ?p1 = <hasWebsite>).
```

[6] The Apache Jena framework provides a tool that heuristically proposes, according to a selectivity estimation, in which order the triples of a SPARQL query should be executed. This is also useful in our context.

In this query we have introduced special variables (`?o1`, `?s1`, `?p1`) and added `FILTER` expressions that list the grounding options for each of them. The results of this query contain those of our 2 queries (up to assignments to the special variables), as well as queries with additional element combinations such as `<Cole_Umbria> <hasWebsite> ?x`.

6 Procuring Feedback

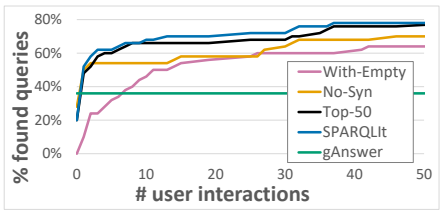
The provenance of candidate queries that were not pruned in earlier stages and example query result (if exists) are returned to the GUI and presented to the user. The user is then prompted for feedback on each triplet in the provenance, and may choose one of the following responses for a given triplet (e, e', v).

- `MUST`: 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 are imposed on the triplet.

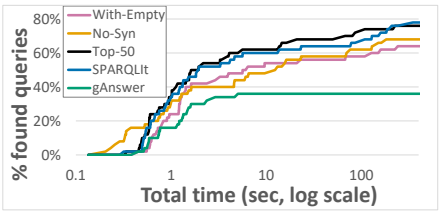
The feedback obtained in each step is accumulated and the corresponding constraints that it entails on the search space of queries are stored by the Constraints Manager. The Groundings Generator then only generates groundings that are valid according to the constraints, and the Triple Store Manager prunes queries whose results do not comply with the feedback.

Example 6.1. *The bottom pane of the screenshots in Figure 2 show triples of original element, proposed element and example variable binding, corresponding to our overall query provenance. Figure 2a shows our running example same semi-formal query alongside a proposed formal query that we shall call Q_6 (on the top-right, where we also have a counter of the number of user interactions done so far). As in Example 3.3, each element of the semi-formal query is mapped to an element of Q_6. Here, the user gave a `MUST` feedback to the mappings of `lives_in`, `columbia`, `has_graduate` (marking $\vee$ in the checkbox on the left-most column on the bottom pane). The user rejected (by marking an X) the elements `dealsWith` and `?s2`$\mapsto$`Canada`, which means that for queries based on the same semi-formal query (3 triples, a two-step path between the person `?x` and `India`) this predicate and query result should not reoccur. The mapping of `?x` remains a `MAYBE` because the user does not know if this person is included in the correct answer or not.*

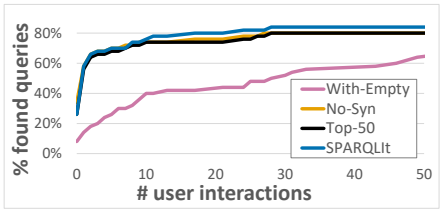
Both positive and negative feedback on Q_6 significantly narrow the search space of possible queries. As a result, and accounting for user feedback from previous iterations, the formal query in Figure 2b is proposed by the system in the next iteration. Indeed, `dealsWith` and `Canada` do not appear here, and the elements marked with `MUST` remain from Q_6 (automatically marked with a $\vee$ by the GUI).



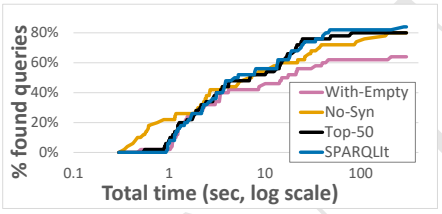
(a) Success by no. interactions (DBpedia)



(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

7 Implementation and Experimental Study

We next describe our experimental study, starting with the implementation and experimental setup, and then providing the experimental results.

7.1 Implementation

We have implemented our solution in a prototype called SPARQLIt. The prototype is implemented in .Net, using Blazor[7] for its front-end, Elasticsearch for the search engine used by the Grounding Generator (recall Section 4) and Apache Jena for the Triple Store (recall Section 5. In our computation of grounding edit distance we have used Levenstein edit distance from the input element and, in the case of predicate, the minimum edit distance from a synonym of this predicate. Synonyms were taken from Datamuse.[8]

Recall the GUI screenshots in Figure 2. So far, we have described the main flow of interaction with the user: the user enters a semi-formal query (top-left panel), and then iteratively receives formal query proposals and gives feedback to refine the proposals (check-boxes on the bottom left). Our implementation further supports additional user actions via the icon buttons on the top panels. From left to right, these buttons stand for: (i) *Configuration*: allows the user to set system parameters such as k, the number of results retrieved from the search engine. (ii) *Reset*: returns the users to their initial query canceling all interactions. (iii) *Undo*: returns to the previous interaction step, by removing the constraints derived from the last user feedback and returning to the formerly proposed formal query. (iv) *Cancel*: terminates the current search execution, to allow the user to change their feedback. (v) *Full Results*: displays the full results of the current formal query proposal (as opposed to the bottom pane, which only shows one example result; this enables the user to gain further intuition about the query output, e.g., if there were too few or too many results. (vi) *Next*: submits the user feedback and gets the next formal query proposal.

[7]Blazor. `https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor/`
[8]Datamuse. `https://www.datamuse.com/api/`

7.2 Experimental Setup

All of our experiments were executed on Intel i7-core processors with 32GB of RAM.

The following RDF KGs were used in our experiments.

- YAGO [49]. A KG Derived from Wikipedia, WordNet, WikiData, GeoNames, and other data sources. Only English facts were used which consist of approximately 160M triples.
- DBpedia [32]. A KG derived mainly from Wikipedia, which consists of approximately 363M triples. By default, we show results for this KG.

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 [52]. Their translation to gold formal queries w.r.t. DBpedia is given in [52], and we have formulated the gold queries w.r.t. YAGO; we have stripped aggregation to obtain selection queries, see the list of used queries in our code repository.[9] We have then constructed three benchmarks of semi-formal queries

- *QALD Translated* is based on manual translation that is oblivious to the terminology used in the KGs, 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`;
- *QALD Cross-KG* uses the formal query for YAGO as a semi-formal query to be evaluated with respect to DBpedia, and vice versa.
- *A synthetic dataset* that includes semi-formal queries that vary different aspects of the query structure (e.g., number of edits needed to obtain the formal query), to examine their effect on our solutions.

Intuitively, *QALD Translated* includes semi-formal queries that depend only on the natural language queries, and hence are independent from any KG, but may depend on the query writer's subjective interpretation. *QALD Cross-KG* queries are not affected by subjective judgement, since their structure is dictated by one of the KGs; but since they are formal queries on one KG, they can only be used as semi-formal input for the other KG. We have tested all benchmarks on both KGs, simulating user feedback using the underlying, hidden, formal query.

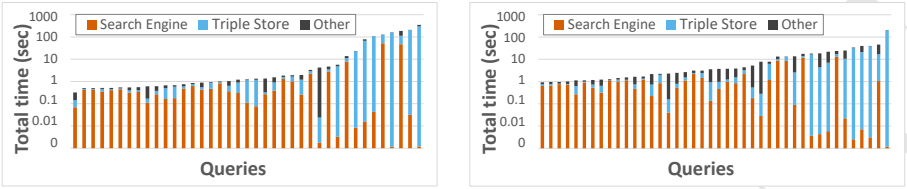
As solution baselines we have used the both a natural language query engine, and multiple variants of SPARQLIt, as follows.

- The NL-to-SPARQL query engine *gAnswer* [23], which achieved the best results in the QALD-9 Challenge [52].
- *With-Empty.* A variant that does not prune queries with empty results.
- *No-Syn.* This variant does not use synonyms for distance computation, unlike our standard implementation (see Section 4).

[9] Code and Query Repository for SPARQLIt: `https://github.com/ycallen/dexa22`

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

Table 3: Percent of successfully found QALD queries



(a) DBpedia (b) YAGO

Figure 4: Time segmentation for SPARQLIt

- *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.

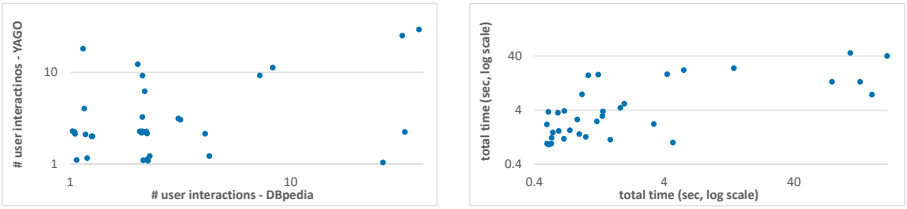
Our evaluation metrics are the *number of user interactions*, i.e., the number of executions of step 4 in Figure 1; and the *total response time*, i.e., the total computation time of proposed queries, throughout the interactive session.

7.3 Experimental Results

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. For both metrics, SPARQLIt exhibited the best results. 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 KG. 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 3 and contrast them with With-Empty and gAnswer (columns 1, 3 and 4 respectively).

We have executed the above experiments using the QALD Cross-KG workload (where YAGO queries are used as semi-formal queries over DBpedia and vice versa). The trends were similar: with one interaction, we have successfully found 44% of the queries in both KGs; 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



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

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

parameters	*#interactions*	*total time*	*#triples*	*#variables*
#interactions	1			
total time	-0.07	1		
#triples	-0.16	0.23	1	
#variables	0.04	-0.05	**0.65**	1
#structural edits	0.22	-0.06	**0.48**	**0.52**

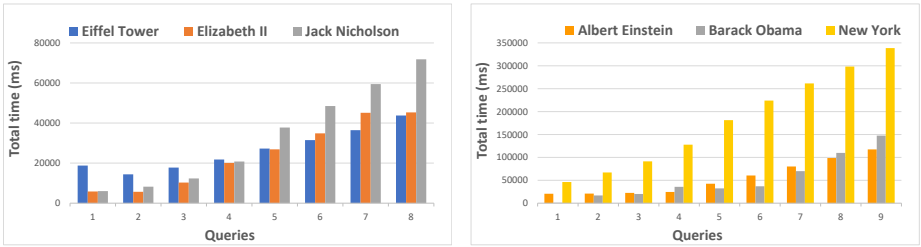
Table 4: Correlation Matrix

summarize the success rates with up to 50 interactions in Table 3 (second column). *Overall, SPARQLIt had a high success rate and has succeeded in finding formal queries for the same semi-formal input over different KGs, 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 the Search Engine, Triple Store and 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 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 KG. Figures 5a and 5b examine the effect of the KG 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 KG structure are indeed essential when writing formal queries.

We have also checked correlation between different parameters of the interactive process for QALD queries over DBpedia, including (1) the number of user interactions;(2) total response time; (3) number of triples in the output formal query; (4) number of distinct variables in the output formal query; (5) number of structural edits performed by SPARQLIt in the background.The results are displayed in the matrix table of Table 4. The highest correlation is observed between query properties (number of variables versus triples). We also observe that the number of performed structural edits is correlated with the numbers



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

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

of triples and variables, since indeed when there are more triples and variables, there are more edits that can be computed and examined. However, note the weak or no correlation between these parameters and the main performance metrics, namely, the total execution time and number of interactions. This suggests that the query difficulty is mainly determined by deeper semantics of the KG, rather than structural aspects of the query.

Synthetic Queries. Figure 6 shows the response time of SPARQLIt for representative synthetic queries (in milliseconds). The synthetic 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.

8 Related Work

There are many lines of work on algorithms and systems that assist in query formulation for different languages, including SPARQL, and interactive solutions that allow for gradual refinement of the query. We next elaborate on ones most related to our setting.

Natural language interfaces. A vast body of work has been dedicated to interfaces that receive as input an unstructured natural language question and translate it into a structured query to execute against a KG (e.g., [9, 14, 16, 23, 43, 54, 56, 61]) or a relational database (e.g., [8, 10, 24, 25, 30, 33, 47, 42]), by using either rule based, statistical, machine learning or deep learning techniques. Compared to the NL approach, our solution requires users to provide a more structured specification in a semi-formal query; however, as we show in our paper, and as indicated in other previous work [60, 43] this structure is helpful. In particular, the recent [43] proposed a deep neutral approach to question answering by first translating the natural language question to a silhouette SPARQL query, and then matching this query to the KG. In our case, the interactive traversal of candidate queries is guided in a principled way by the transformations performed on the semi-formal query, hence it is eminent to our interactive solution (see Section 7). Another shortcoming of some NL interfaces is that the solution typically requires expensive training data (in the case

of ML solutions with supervision) or manual rule specification, and is therefore not easily transferable from one KG to another.

Interactive Querying and Information Retrieval. Another research field related to ours is that of interactive querying/information retrieval, which has been studied from different perspectives. One line of research studies the translation NL questions or searches to queries in the context of previous questions [26, 38, 50, 57]. While resembling our work in allowing for interactivity, this line of work focuses on the interpretation of questions based on the context, and not on an interactive search for the right query. In our case, the interaction focuses on feedback from the user, which points out what parts of the proposed query were not correctly captured. Another line of work considers guiding information retrieval or query reformulation via ontological information [20, 48]. This work mostly focuses on selecting terms, adding related terms or resolving the ambiguity of terms based on domain knowledge recorded in an ontology. A crucial difference of our setting with respect to this work is the use of semi-formal queries, allowing to capture, on the one hand, structural properties of the user intent, but on the other hand allowing to manipulate this structure according to the queried KG.

Keyword search. This line of work considers the finding of sub-graphs in the KG containing all or some of given query keywords and formulating a query that captures these sub-graphs (e.g., [21, 28, 29, 34]) which was also considered for SQL interfaces [3, 51]. This approach has common properties with the above mentioned NL approach, as it is appealing to novice users. Compared with the NL approach, solutions here typically rely on the sub-graph structure and hence do not require expensive training. Again, a major challenge is recovering from a situation where no suitable query was found, i.e., how to make the process interactive.

Auto-completion and visual interfaces. Different interfaces allow users to type SPARQL queries and automatically receive proposals for the potential values, elements or query parts that match the partial user input, by typing, selecting from lists or from visual elements (e.g., [12, 17, 44, 41, 45, 31, 55]). The work of [17] further propose corrections to the formed query, such that these corrections may for instance increase the number of query results. The drawback of this approach is that KGs are typically very sparse and noisy. Hence, there may exist many real KG elements/query snippets that superficially seem relevant but actually do not match the real user intention or the other parts of the query. In contrast, our approach (i) simultaneously searches for all the parts of the user-specified query, such that we prefer candidate formal queries that match all the properties requested by the user; and (ii) through their feedback on both query parts and results, users can browse additional candidate queries without reaching a "dead-end". We may view this approach as complementary to ours: the users may use auto-completion or a visual interface to guide their editing of a semi-formal query, and then use our approach to interactively browse through related queries and find ones that return the intended results.

Faceted browsing. 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, 19, 22, 37, 53]). The challenge for interaction in this context is similar to the above mentioned auto-completion and visual interfaces: when the browsed query parts 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.

Semantic similarity search and query reformulation. Similarity search is a means of extending or reformulating an initial query with similar queries that return additional results (e.g., [18, 39, 40, 46, 58, 63, 64]. In particular, Zheng et al. [63] studied semantic similarity search for SPARQL, and introduced an edit distance notion for RDF graphs. The edit 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.

Query by example. This area of research studies the "reverse-engineering" of queries based on positive/negative result examples provided by the use (e.g., [1, 2, 11, 6, 13, 15, 27, 35, 59], see [36] for a comprehensive survey). In RDF KGs, this method can be effective when the users search categories or classes, for which they can easily provide positive and negative examples. The method becomes challenging to use when the query includes non-categorical predicates, which are typically very sparse and heterogeneous, or when users cannot provide sufficient examples.

9 Conclusion

In this paper we introduced SPARQLIt, a framework that helps users querying RDF knowledge graphs, by allowing them to construct semi-formal SPARQL queries that have a SPARQL syntax but do not necessarily match in contents and structure to the graph, and hence do not require prior knowledge of the latter. Provenance is used to track the transformation of the input query to the candidate formal query, to give the user an intuition on the correspondence between the parts of the input query and the candidate query, and to allow them to specify fine-grained feedback at the level of query parts and variable bindings. This yields iterative and interactive process of adjusting the query proposals to the user intentions. Our experimental study, conducted on real knowledge graphs using the QALD benchmark for queries indicate that our approach is practical in terms of user effort and latency, and that very few interactions suffice to outperform a state-of-the-art natural language interface for SPARQL.

For future research, recall that we have listed 3 operators used by the Structural Edits Generators. Our framework design is flexible in this sense and allows adding or changing operators. It would be interesting, given logs of queries to SPARQL endpoints and/or logs of interactions with SPARQLIt, to user machine learning or data mining to discover useful operators, or weights for the next operators given a semi-formal query. Moreover, we may consider extending the fragment of SPARQL we have focused on, to allow for, e.g., group-by, aggregation, difference, etc. While we believe that fining selection queries solves

the main challenge in querying an unknown KG, it would be then interesting, e.g., to combine methods for interactive data science (e.g., [62]) to precise the query operators. Another challenging direction is considering RDF inference rules, specified by languages such as OWL.[10] In the present work, inference rules are implicitly accounted for by the Triple Store, which uses inference in computing query answers. However, we may consider explicitly accounting for such inferences, thereby, e.g., pruning more candidate groundings.

Acknowledgments 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 article is published in Springer Knowledge and Information Systems, and is available online at `https://doi.org/10.1007/s10115-023-01939-x`. An official view-only version is freely available at `https://rdcu.be/dl53C`.

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*, pages 49–60, 2013.

[2] E. Abramovitz, D. Deutch, and A. Gilad. Interactive inference of SPARQL queries using provenance. In *ICDE*, pages 581–592, 2018.

[3] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, Parag, and S. Sudarshan. BANKS: browsing and keyword searching in relational databases. In *PVLDB*, pages 1083–1086, 2002.

[4] Y. Amsterdamer and Y. Callen. SPARQLIt: Interactive SPARQL query refinement. In *ICDE*, pages 2649–2652, 2021.

[5] Y. Amsterdamer and Y. Callen. Provenance-based SPARQL query formulation. In *DEXA*, volume 13426, pages 116–129, 2022.

[6] M. Arenas, G. I. Diaz, and E. V. Kostylev. Reverse engineering SPARQL queries. In *WWW*, pages 239–249, 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*, pages 123–126, 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, 2019.

[9] N. Bhutani, X. Zheng, and H. V. Jagadish. Learning to answer complex questions over knowledge bases with query composition. In *CIKM*, pages 739–748, 2019.

[10] OWL 2 Web Ontology Language Primer (Second Edition). `https://www.w3.org/TR/2012/REC-owl2-primer-20121211`

[10] L. Blunschi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger. SODA: generating SQL for business users. *PVLDB*, 5(10):932–943, 2012.

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

[12] S. Dastgheib, D. I. McSkimming, K. Natarajan, and K. J. Kochut. Sparqling: A graphical interface for SPARQL. In *ISWC*, volume 1486, 2015.

[13] G. I. Diaz, M. Arenas, and M. Benedikt. SPARQLByE: Querying RDF data by example. *PVLDB*, 9(13):1533–1536, 2016.

[14] D. Diefenbach, K. D. Singh, and P. Maret. WDAqua-core1: A question answering service for RDF knowledge bases. In *WWW comp.*, pages 1087–1091, 2018.

[15] K. Dimitriadou, O. Papaemmanouil, and Y. Diao. Explore-by-example: an automatic query steering framework for interactive data exploration. In *SIGMOD*, pages 517–528, 2014.

[16] M. Dubey, S. Dasgupta, A. Sharma, K. Höffner, and J. Lehmann. AskNow: A framework for natural language query formalization in SPARQL. In *ESWC*, pages 300–316, 2016.

[17] A. El-Roby, K. Ammar, A. Aboulnaga, and J. Lin. Sapphire: Querying RDF data made simple. *PVLDB*, 9(13):1481–1484, 2016.

[18] S. Elbassuoni, M. Ramanath, and G. Weikum. Query relaxation for entity-relationship search. In *ESWC*, pages 62–76, 2011.

[19] S. Ferré. Expressive and scalable query-based faceted search over SPARQL endpoints. In *ISWC*, pages 438–453, 2014.

[20] E. García-Barriocanal and M. Á. S. Urbán. Designing ontology-based interactive information retrieval interfaces. In R. Meersman and Z. Tari, editors, *OTM*, volume 2889, pages 152–165, 2003.

[21] K. Golenberg and Y. Sagiv. A practically efficient algorithm for generating answers to keyword search over data graphs. In *ICDT*, pages 23:1–23:17, 2016.

[22] F. Haag, S. Lohmann, S. Siek, and T. Ertl. QueryVOWL: Visual composition of SPARQL queries. In *ESWC*, pages 62–66, 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):824–837, 2018.

[24] P. Huang, C. Wang, R. Singh, W. Yih, and X. He. Natural language to structured query generation via meta-learning. In *NAACL-HLT*, pages 732–738, 2018.

[25] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. Learning a neural semantic parser from user feedback. In *ACL*, pages 963–973, 2017.

[26] M. Iyyer, W. Yih, and M. Chang. Search-based neural structured learning for sequential question answering. In *ACL*, pages 1821–1831, 2017.

[27] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri. Querying knowledge graphs by example entity tuples. In *ICDE*, pages 1494–1495, 2016.

[28] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *PVLDB*, pages 505–516, 2005.

[29] G. Kadilierakis, P. Fafalios, P. Papadakos, and Y. Tzitzikas. Keyword search over RDF using document-centric information retrieval systems. In *ESWC*, volume 12123, pages 121–137, 2020.

[30] H. Kim, B. So, W. Han, and H. Lee. Natural language to SQL: where are we today? *PVLDB*, 13(10):1737–1750, 2020.

[31] V. Kritsotakis, Y. Roussakis, T. Patkos, and M. Theodoridou. Assistive query building for semantic data. In *SEMANTiCS*, volume 2198, 2018.

[32] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. Dbpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[33] F. Li and H. V. Jagadish. NaLIR: an interactive natural language interface for querying relational databases. In *SIGMOD*, pages 709–712, 2014.

[34] X. Lin, Z. Ma, and L. Yan. RDF keyword search using a type-based summary. *J. Inf. Sci. Eng.*, 34(2):489–504, 2018.

[35] M. Lissandrini, K. Hose, and T. B. Pedersen. Example-driven exploratory analytics over knowledge graphs. In *EDBT*, pages 105–117, 2023.

[36] M. Lissandrini, D. Mottin, T. Palpanas, and Y. Velegrakis. *Data Exploration Using Example-Based Methods*. Synthesis Lectures on Data Management. 2018.

[37] M. Lissandrini, D. Mottin, T. Palpanas, and Y. Velegrakis. Graph-query suggestions for knowledge graph exploration. In *WWW*, pages 2549–2555, 2020.

[38] Q. Liu, B. Chen, J. Lou, G. Jin, and D. Zhang. FANDA: A novel approach to perform follow-up query analysis. In *AAAI*, pages 6770–6777, 2019.

[39] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: Give me an example of what you need. *PVLDB*, 7(5):365–376, 2014.

[40] K. Munir, M. Odeh, and R. McClatchey. Ontology-driven relational query formulation using the semantic and assertional capabilities of OWL-DL. *Knowl. Based Syst.*, 35:144–159, 2012.

[41] M. Papadaki, N. Spyratos, and Y. Tzitzikas. Towards interactive analytics over RDF graphs. *Algorithms*, 14(2):34, 2021.

[42] H. Poon. Grounded unsupervised semantic parsing. In *ACL*, pages 933–943, 2013.

[43] S. Purkayastha, S. Dana, D. Garg, D. Khandelwal, and G. P. S. Bhargav. A deep neural approach to KGQA via SPARQL silhouette generation. In *IJCNN*, pages 1–8, 2022.

[44] K. Rafes, S. Abiteboul, S. C. Boulakia, and B. Rance. Designing scientific SPARQL queries using autocompletion by snippets. In *eScience*, pages 234–244, 2018.

[45] S. Sana e Zainab, M. Saleem, Q. Mehmood, D. Zehra, S. Decker, and A. Hasnain. FedViz: A visual interface for SPARQL queries formulation and execution. In *VOILA@ISWC*, volume 1456, page 49.

[46] R. Schenkel, A. Theobald, and G. Weikum. Semantic similarity search on semistructured data with the XXL search engine. *Inf. Retr.*, 8(4):521–545, 2005.

[47] 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):2747–2759, 2020.

[48] N. Stojanovic, R. Studer, and L. Stojanovic. An approach for step-by-step query refinement in the ontology-based information retrieval. In *WI*, pages 36–43, 2004.

[49] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A Core of Semantic Knowledge. In *WWW*, pages 697–706, 2007.

[50] A. Suhr, S. Iyer, and Y. Artzi. Learning to map context-dependent sentences to executable formal queries. In *NAACL-HLT*, pages 2238–2249, 2018.

[51] S. Tata and G. M. Lohman. SQAK: doing more with keywords. In *SIGMOD*, pages 889–902, 2008.

[52] R. Usbeck, R. H. Gusmita, A. N. Ngomo, and M. Saleem. 9th challenge on question answering over linked data (QALD-9). In *ISWC*, pages 58–64, 2018.

[53] H. Vargas, C. B. Aranda, A. Hogan, and C. López. RDF Explorer: A visual SPARQL query builder. In *ISWC*, volume 11778, pages 647–663, 2019.

[54] D. Vollmers, R. Jalota, D. Moussallem, H. Topiwala, A. N. Ngomo, and R. Usbeck. Knowledge graph question answering using graph-pattern isomorphism. 53:103–117, 2021.

[55] D. Vrandecic and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.

[56] S. Walter, C. Unger, P. Cimiano, and D. Bär. Evaluation of a layered approach to question answering over linked data. In *ISWC*, volume 7650, pages 362–374, 2012.

[57] X. Wang, S. Wu, L. Shou, and K. Chen. An interactive NL2SQL approach with reuse strategy. In *DASFAA*, volume 12682, pages 280–288, 2021.

[58] 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*, pages 445–456, 2020.

[59] Y. Y. Weiss and S. Cohen. Reverse engineering SPJ-queries from examples. In *PODS*, pages 151–166, 2017.

[60] 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*, 8:183–198, 2020.

[61] X. Yin, D. Gromann, and S. Rudolph. Neural machine translating from natural language to SPARQL. *Future Gener. Comput. Syst.*, 117:510–519, 2021.

[62] Y. Zhang and Z. G. Ives. Finding related tables in data lakes for interactive data science. In *SIGMOD*, pages 1951–1966, 2020.

[63] W. Zheng, L. Zou, W. Peng, X. Yan, S. Song, and D. Zhao. Semantic SPARQL similarity search over RDF knowledge graphs. *PVLDB*, 9(11):840–851, 2016.

[64] G. Zhu and C. A. Iglesias. Sematch: Semantic similarity framework for knowledge graphs. *Knowl. Based Syst.*, 130:30–32, 2017.