Interactive Inference of SPARQL Queries Using Provenance

Efrat Abramovitz
School of Computer Science
Tel Aviv University
Tel Aviv, Israel
abramovitz2@mail.tau.ac.il

Daniel Deutch
School of Computer Science
Tel Aviv University
Tel Aviv, Israel
danielde@post.tau.ac.il

Amir Gilad
School of Computer Science
Tel Aviv University
Tel Aviv, Israel
amirgilad@mail.tau.ac.il

***Abstract*—Inference of queries from their output examples has been extensively studied in multiple contexts as means to ease the formulation of queries. In this paper we propose a novel approach for the problem, based on provenance. The idea is to use provenance in two manners: first as an additional information that is associated with the given examples and explains their rationale; and then again as a way to show users a description of the difference between candidate queries, prompting their feedback. We have implemented the framework in the context of simple graph patterns and union thereof, and demonstrate its effectiveness in the context of multiple ontologies.**

I. Introduction

SPARQL is the predominant query language for ontology querying, allowing users to fetch, filter, and manipulate data. For a given query, *provenance* information is a form of meta-data that is associated with query results, to describe the origin of each piece of data and the computational process leading to its generation. The tracking, storage and presentation of provenance have been extensively studied in general [1], [2], [3], [4], [5], [6], [7], and specifically in the context of SPARQL [8], [9].

The main contribution of this paper is a novel framework, that uses provenance for the *inference* of SPARQL queries. The high-level idea is to leverage provenance information in two ways, as follows.

First, provenance is provided as an input to the framework. In this sense, our solution resembles the lines of work on query-by-example and query-by-output [10], [11], [12], [13], [14], with our novelty being in that we attach provenance information to each given output example. This provenance information could either come from a recorded trace of the evaluation of the query (where the query itself has been lost, the latter being the assumption in e.g. [11]), or compiled from an *explanation* provided by the user. The provenance model that we use is simple enough – namely the ontology sub-graph which is the image of the query – so that intuitively formulated explanations may be compiled to it.

Providing provenance information along with the given examples naturally narrows the search space, but both our theoretical analysis and experimental results indicate that it may still be far from uniquely dictating a query. Here is where our second use of provenance comes into play, as follows. We first restrict the attention to k queries, by requiring a match to the explanations, combined with a ranking and pruning mechanism. Then we repeatedly show an *explanation* of the difference between each two queries – namely an output of one query that is not an output of the second query, *along with its provenance with respect to the first query*. Here again, the general interactive approach has already been employed for data exploration [15], but the novelty here is again our use of provenance, this time as means of explaining differences between candidate queries.

As a simple illustrative example, consider an ontology of authors and papers, and a query asking for all authors with Erdős number 2. Examples for the query output, i.e. example authors, will likely be un-indicative of the actual intended query, since the authors may share many other characteristics. The *provenance* of an example author with respect to the intended query will be one of her co-authorship paths to Erdős of length 2. This additional information clearly focuses the search for a query, but will still allow for multiple queries in addition to the intended one, for instance one that uses a constant for the name of the third author in the chain if this name was the same for the given examples. In such cases, showing not only an example of the difference between the query results (i.e. an author with Erdős number 2 through a different "middle" author), but rather the provenance of such results, i.e. the chain going through the other middle author, will allow users to distinguish between the queries.

Towards a solution, several modeling and algorithmic challenges are addressed, as follows.

Model and Problem Formulation (Section II): We first need to define the provenance model and the formal notion of a query that is consistent with a given provenance information. We focus in this paper on a simple class of SPARQL queries, namely basic graph patterns with a single output node and union thereof, possibly with disequalities. For such queries there is an intuitive notion of provenance for a given result node n – the ontology subgraph that is the image of some homomorphism from the query to the ontology graph, which yields n. Then, a query is consistent with a set of examples and their provenance information if evaluating the query yields each of the examples with provenance that is isomorphic to the given one.

Simple Queries (Section III): We then study the problem of finding consistent queries, starting by assuming that the target query consists of a single pattern (i.e. no union). We show that it is trivial to find some consistent query, but there are many such queries, and some are more natural than others. To this effect, we devise a simple preference criterion, namely that of the number of variables occurring in the query. Intuitively there is a correlation between the number of variables in the query and the "tightness" of its fit to the given examples. We show that it is NP-hard to find one that is a "tight fit" in the sense that its number of variables is minimal. We further devise a greedy heuristic solution aimed at minimizing the number of variables.

Union Queries (Section IV): When union is allowed, there is always a trivial query with no variables whose simple patterns are precisely those given as explanations to examples. There is a trade-off between the number of patterns and the number of variables in each pattern – intuitively when we merge two explanations into a single pattern, we may need to introduce variables to account for different labels in the two explanations. Our target then is to minimize some weighted average of the two desiderata, and again we apply a greedy solution. We further provide a natural greedy extension that maintains at each step not only the current best query but rather the top-k queries in this step.

Feedback (Section V): As mentioned above, we use provenance not only as the input to the system but also to prompt user feedback that allows us to choose between the candidate queries. We show how to efficiently generate provenance for the output of a difference query between two candidates; this provenance information is then presented to users and their answer is in turn used to disqualify candidate queries. This is done repeatedly to focus on the intended query.

Implementation and Experiments (Section VI): We have implemented our framework and have conducted experiments using three different ontologies and query workloads. We have conducted automated experiments, in which we sample output examples and their provenance and examine whether we could reverse-engineer the underlying queries depending on the number of given examples. We have also asked users to provide examples and explanations that match either queries that were given to them or ones of their own. The results indicate the feasibility of our approach, as our implementation was able in most cases to reconstruct the query based on a fairly small number of examples.

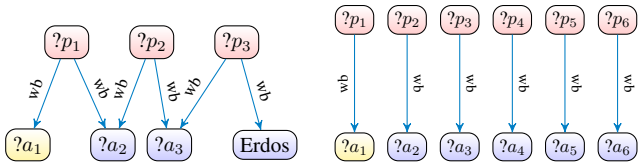
We overview related work in Section VII and conclude in Section VIII. For lack of space, some of the proofs are deferred to the full version [16].

II. PRELIMINARIES AND MODEL

We now discuss the necessary SPARQL background including ontology graphs and also define our provenance model.

A. Graph Model

For data representation, we assume infinite sets of predicates $\mathcal{P}$ and of values $\mathcal{V}$. An ontology database is a graph $\mathcal{O} = (V, E, L_E, L_V)$, where $L_E : E \to \mathcal{P}$ and $L_V : V \to \mathcal{V}$ are functions mapping the edges and nodes to predicates and values, respectively. L_v is required to be one-to-one, i.e., there is at most one node in the ontology with each value. There may be multiple edges between each two nodes but their predicates must then be distinct. As a running example we will use the small ontology of publications depicted in Figure 1a. All edges are labeled wb ("written by") and each node is mapped to a unique value appearing in it.



(a) Q_1 (b) Q_2

Fig. 2: Consistent Queries

To define queries, we extend the range of the function L_V to further include an infinite set of variable names $Vars$. A simple SPARQL query is then a tuple (V, E, L_E, L_V, v) where V, E, L_E are defined as before, $L_V : V \to \mathcal{V} \cup Vars$, and $v \in V$, for which $L_V(v) \in Vars$, is called the projected node.

Example 2.1: Examples for simple SPARQL queries are depicted in Figure 2 where $?a_1$ is the projected node.

The semantics of queries is that we look for mappings from the query to some subgraph of the ontology, and for each such mapping, output the node to which the projected node was mapped. This is formalized by the next definition of a *match*.

Definition 2.2: Given a simple SPARQL query Q and an ontology graph $\mathcal{O}$, a match of Q to $\mathcal{O}$ is two functions $\mu_1 : V(Q) \to V(\mathcal{O})$ and $\mu_2 : E(Q) \to E(\mathcal{O})$ such that (1) if $\mu_1(u) = v$ and $L_V(u) \in \mathcal{V}$ then $L_V(u) = L_V(v)$, (2) if $\mu_1(u) = v$ and $L_V(u) \in Vars$ then for each w such that $L_V(u) = L_V(w)$, it holds that $L_V(\mu_1(u)) = L_V(\mu_1(w))$, (3) if $\mu_2(e) = e'$ then $L_E(e) = L_E(e')$, and (4) if $\mu_2(e) = e'$ where $e = (u, v), e' = (w, x)$ then $\mu_1(u) = w, \mu_1(v) = x$.

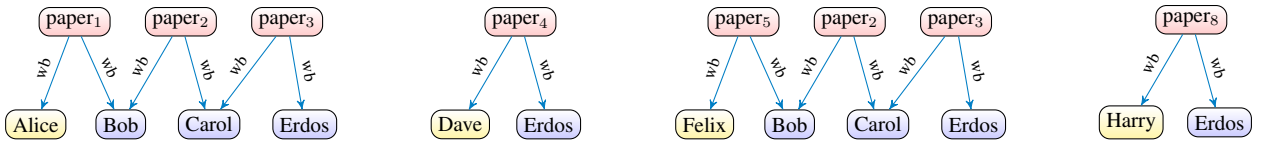
Abusing notation, we will sometimes use a single notation μ for a mapping of both nodes and edges. Let $v \in V(Q)$ be the projected node of a query Q and let μ be such a mapping. Then $\mu(v)$ is a result of Q defined by μ and is denoted by $Q_\mu(\mathcal{O})$. Since a query Q may have several matches to an ontology $\mathcal{O}$, yielding several results, the evaluation result of the query Q w.r.t the ontology $\mathcal{O}$ is defined as $Q(\mathcal{O}) = \bigcup_\mu Q_\mu(\mathcal{O})$.

Example 2.3: Consider the ontology E_1 depicted in Figure 1a, and the query Q_1 depicted in Figure 2a. Q_1 matches the subgraph since we can define a mapping μ by $\mu(?p_1) = paper_1$, $\mu(?p_2) = paper_2$, $\mu(?p_3) = paper_3$, $\mu(?a_1) = Alice$, $\mu(?a_2) = Bob$, $\mu(?a_3) = Carol$, $\mu(\text{Erdős}) = Erdos$. The output $Q_{1_\mu}(E_1)$ is the value $Alice$, and this is also $Q_1(E_1)$.

A SPARQL query is a collection of simple SPARQL queries $Q = Union(\{q_1, \ldots, q_n\})$, where the output is defined to be the union of the output sets, i.e. $Q(\mathcal{O}) = q_1(\mathcal{O}) \cup \ldots \cup q_n(\mathcal{O})$.

B. Graph Provenance

SPARQL provenance can be described as a graph, relying



(a) E_1 (b) E_2 (c) E_3 (d) E_4

Fig. 1: Explanations

on the graph structure of the ontology and query and the match between them (Definition 2.2).

Definition 2.4: For a given SPARQL simple query Q, an ontology graph $\mathcal{O}$, and a match μ of Q to $\mathcal{O}$ yielding the result res, the provenance of res with respect to μ is $prov_\mu(res) = \mu(Q)$, i.e., the image of μ for Q.

Combining all matches of Q to $\mathcal{O}$ yielding the result res gives the provenance of res, defined as
$prov(res) = \{\mu(Q) \mid \mu \text{ is a matching yielding res}\}$.

The above definition intuitively captures provenance as a set of graphs whose nodes and edges are the images of matches of the query Q to the ontology $\mathcal{O}$.

Given a SPARQL query $Q = Union(\{q_1, \ldots, q_n\})$, an ontology graph $\mathcal{O}$, and a result res, the provenance of res is $prov(res) = prov_{q_1}(res) \cup \ldots \cup prov_{q_n}(res)$, where $prov_{q_i}(res)$ is the provenance of res w.r.t the query q_i and the ontology $\mathcal{O}$.

We now define the problem of learning queries from example results and their explanations. We first introduce the notion of explanations:

Definition 2.5: Let $\mathcal{O}$ be an ontology. An explanation is a subgraph of $\mathcal{O}$, i.e., $E \subseteq \mathcal{O}$ that has a node $v \in V(E)$ called a distinguished node and denoted by $dis(E)$.

Intuitively, the distinguished node is the output example the user expects and the rest of the explanation describes the reason that the user has chosen it. The same distinguished node may appear in multiple explanations. We call a set of explanations an *e*xample-set.

Our goal is to create a query that will output every distinguished node in the example-set, with the explanation of that distinguished node matching at least one of the graphs in the provenance of that node. We next formally define the notion of a query being *consistent* with an example-set.

Definition 2.6: Given an ontology graph $\mathcal{O}$, an example-set Ex, and a SPARQL query Q we say that Q is consistent with respect to Ex if for every $E \in Ex$ and its distinguished node res it holds that $res \in Q(\mathcal{O})$, and E is isomorphic to some G in $prov_Q(res)$, i.e. the provenance of res w.r.t Q and $\mathcal{O}$ contains E. *CONSISTENT-QUERY* is the problem of finding a consistent query for a given example-set.

We next demonstrate the notion of consistent queries with respect to a given example-set.

Example 2.7: Consider the example defined by E_1, E_2, E_3, E_4 presented in Figure 1, and their output nodes $Alice$, $Dave$, $Felix$, and $Harry$. Further consider Q_1 depicted in Figure 2a. There is a match μ_1 between Q_1 and E_1 producing $Alice$ detailed in Example 2.3. In a similar manner, there is a match between Q_1 and E_2, E_3, E_4, thus Q_1 is consistent with the example-set. Q_2 is also consistent since there is are matches from Q_2 to each of the explanations. In particular, the match $\mu_2 : Q_2 \to E_1$ is defined by $\mu_2(?a_1) = \text{Alice}$, $\mu_2(?a_2) = \mu_2(?a_3) = Bob$, $\mu_2(?a_4) = \mu_2(?a_5) = Carol$, $\mu_2(?a_6) = \text{Erdős}$, $\mu_2(?p_1) = \mu_2(?p_2) = paper_1$, $\mu_2(?p_3) = \mu_2(?p_4) = paper_2$, $\mu_2(?p_5) = \mu_2(?p_6) = paper_3$.

III. Simple Queries

We start by analyzing the problem of CONSISTENT-QUERY for simple queries.

We first note that given a candidate query and an example-set, verifying consistency amounts to deciding the existence of an *onto* homomorphism from the query to each of the explanations in the example-set. Intuitively, the homomorphism is required to be onto, since it means that all parts of each given explanation appear in the provenance. Performing this verification is NP-complete in the size of the query [17].

Interestingly, this does not imply hardness of the CONSISTENT-QUERY problem. In fact, we can efficiently *generate* a consistent query if one exists, and otherwise declare that it does not.

Proposition 3.1: Given an example-set, there is a PTIME algorithm that decides whether a consistent simple query exists and if it does, returns it.

Proof: To prove the claim, we describe the polynomial algorithm and prove its correctness. First, traverse all explanations and check whether the *set* of edge labels in all explanations is identical. If there is an explanation E_i with an edge label not appearing in another explanation E_j, there is no simple SPARQL query consistent with both. Second, traverse all explanations and check if one of the following holds: (1) the intersection of sets of edge labels whose *source* is the distinguished node is non-empty, or (2) the intersection of sets of edge labels whose *target* is the distinguished node in each explanation is non-empty. We claim that if both (1) and (2) do not hold, no simple consistent query exists:

Lemma 3.2: If the intersection is empty, there is no simple SPARQL query consistent with the example-set.

In both cases mentioned above, the algorithm terminates. Otherwise, start constructing the query Q as follows. For each edge label l, find the explanation with the maximum number m of appearances of an edge with the label l, and add to Q m edges with the label l where each one of these edges has unique variables as their nodes. The projected node x is chosen arbitrarily from the variable nodes that are the source/target of edges with a label appearing in the intersection I (see the reason for this in the sequel). At the end of the process, Q is composed of a set of disjoint edges. For an explanation $E_i \in Ex$, every edge with label l in Q can be mapped to any edge with label l in E_i (the generation process of Q ensures that

their number is sufficient). Moreover, the edge whose source (target) is the projected node can be mapped to the edge with the distinguished node in E_i, since there is at least one edge the same label whose source (target). ■

Example 3.3: Consider the example-set composed of the explanations depicted in Figure 1. The algorithm first checks that the set of labels in E_1 is identical to the set of labels in E_2 and the set of edges connected to the distinguished nodes Alice, Dave, Felix, Harry all have an edge with the label wb connected to them (i.e., the intersection of edges adjacent to the distinguished node in all explanations is non-empty). Since it is, the algorithm constructs a simple query by taking the maximum number of edges in one explanation with each label l. In particular, in our example $l = wb$. Therefore, the algorithm adds 6 edges with the label wb to the query since this is the maximum number of edges with this label in a single explanation in the example (the explanations in Figures 1c, 1d). The algorithm chooses the one of the author nodes arbitrarily to be the projected node. The algorithm then terminates and outputs query Q_2 depicted in Figure 2b.

The query generated by the above process is in general not very "interesting", as it does not capture the connections between the objects described by the user. There may be other consistent queries. For example, consider Q_2 depicted in Figure 2b. An author returned by Q_2 will not necessarily have any connection to Erdős. In general:

Proposition 3.4: There may be exponentially many consistent simple queries with a given example-set, such that no two queries are equivalent.

To guide our search, we thus impose a natural desideratum with respect to the query: that it has a minimal number of variables out of all consistent queries. This intuitively allows to avoid queries that are too general, and instead leads to preferring queries that are more specific to the given examples. In general, fully achieving this desideratum is NP-complete:

Proposition 3.5: Given an example with 2 explanations and an integer k, the problem of deciding whether a simple SPARQL query with at most k variables exists is NP-complete (in the size of the explanations).

In light of this hardness result, we focus on proposing a heuristic solution that generates consistent queries. We start with the case where the input example-set consists of only two explanations, and will then extend the solution to the general case. We define the auxiliary tool of a *complete relation*, which intuitively pairs edges that can later be mapped from a single edge of the generated query.

Definition 3.6: Given explanations E_1, E_2, with edge sets $E(E_1), E(E_2)$, a set $R \subseteq E(E_1) \times E(E_2)$ is a complete relation if the following hold:

1) $\forall (e_1, e_2) \in R.\ L_E(e_1) = L_E(e_2)$
2) $\forall e_1 \in E(E_1)\ \exists e_2 \in E(E_2)$ s.t. $(e_1, e_2) \in R$
3) $\forall e_2 \in E(E_2)\ \exists e_1 \in E(E_1)$ s.t. $(e_1, e_2) \in R$
4) $\exists (e_1, e_2) \in R$ s.t. the sources/targets of e_1 and e_2 are the distinguished nodes of E_1, E_2, respectively.

We show that any consistent queries can stem from some complete relation, using a sequence of basic operations we next define.

Definition 3.7: Given a complete relation R over the explanations E_1, E_2, we say that R leads to a query Q if Q can be formed from R by the following actions:

1) Adding an edge e to Q for every pair in $(e_1, e_2) \in R$ with the label $L_E(e_1) = L_E(e_2)$ and two nodes whose labels are fresh variables.
2) If the sources/targets of e_1, e_2 such that $(e_1, e_2) \in R$ are the distinguished nodes of E_1, E_2, mark the source/target of the edge formed in Q as the projected node. If multiple projected nodes are formed in this process, unify them to a single node.
3) Connecting edges $e_1, e_2 \in E(Q)$ in their source if the pairs leading to their generation $(a_1, b_1), (a_2, b_2) \in R$ satisfy that a_1, a_2 have the same source, and b_1, b_2 have the same source (symmetrically for the target).
4) Set a constant in the source of an edge $e \in E(Q)$ if the pair leading to its generation $(a, b) \in R$ satisfies that the source of a is identical to the source of b (symmetrically for the target).

For the purpose of our following results, operations 1 and 2 are mandatory and operations 3 and 4 can be either performed or ignored for every edge.

Intuitively, operations 1 and 2 lead to a consistent query as shown in Proposition 3.1. Operations 3 and 4 are aimed at minimizing the number of variables in the query but are optional, meaning that they are not necessary for consistency of the formed query. Operation 2 may lead to multiple projected nodes since there may be several pairs of edges in the complete relation whose ends are the distinguished nodes, giving rise to the need to unify them into one node. Note that if this is the case, the first (second) edge in any such pair is connected to all the other first (second) edges in the corresponding pairs in the explanation, so unifying the projected nodes formed in this process into a single node in the query maintains its consistency. We now have:

Proposition 3.8: Given two explanations E_1, E_2, every complete relation R over E_1, E_2 and a sequence of the operations in Definition 3.7 leads to consistent query.

In the other direction:

Proposition 3.9: Given two explanations E_1, E_2, for every consistent query there is a complete relation (Definition 3.6) that can lead to it through a sequence of the operations in Definition 3.7.

Furthermore, we can efficiently generate a query with minimal number of variables *for a given complete relation*:

Proposition 3.10: Given a complete relation, there is a polynomial time algorithm that finds a sequence of operations leading to a consistent simple query with minimum variables (w.r.t the given relation).

Greedy Approach: We thus devise an alternative solution that traverses the space of complete relations in a greedy manner. Intuitively, we want to pair an edge from one explanation with its most similar counterpart from the other explanation, and then create a query edge for each pair. This will make the query edges most "specific" as the query edge

will be mapped to these edges through the matching. To this end, we devise a dynamic gain function for each pair of edges $e_1 \in E(E_1)$ and $e_2 \in E(E_1)$. The gain of choosing each pair of edges is intuitively defined according to the constants in their nodes and based on previously selected pairs of edges, so that previous pairings of e_1 and e_2, and pairings of neighboring edges will be considered as well. The function is essentially a weighted average of the three criteria, depending on e_1, e_2 and possibly the currently generated relation R (which we refer to as a *partial relation*):

1) $c_1(R, e_1, e_2)$ returns 0 is both the sources and targets of e_1 and e_2 have distinct constants 1 if one of them is the same, and 2 if both are the same. (e.g., preferring $e_1 = (a, b), e_2 = (a, c)$ over $e_1 = (d, b), e_2 = (e, c)$)
2) $c_2(R, e_1, e_2)$ returns 0 if both e_1 and e_2 are already paired with some edge in R, returns 1 if one of them was not paired, and 2 if neither of them were paired (e.g., if the pairs $(e_1, e_2), (e_3, e_4)$ were prioritized before (e_1, e_4), then after choosing $(e_1, e_2), (e_3, e_4)$, we would not like to choose (e_1, e_4) since both edges already have better counterparts)
3) $c_3(R, e_1, e_2)$ returns 2 if the sources of e_1, e_2 were also the sources of some $(e_3, e_4) \in R$ and the targets of e_1, e_2 were also the targets of some $(e_5, e_6) \in R$, returns 1 if either the sources or the targets also belong to some $(e_3, e_4) \in R$, and returns 0 if neither the sources nor the targets of e_1, e_2 have been previously paired in R. (e.g., if the pair $((a, b), (c, d))$ was previously chosen, the gain from choosing a pair $((a, e), (c, f))$ increases since we can map the source of both query edges that will be formed from these two pairs to the same variable, thus increasing the number of variables in the future query by only one and not two)

Definition 3.11: Given two explanations, a pair of edges $e_1 \in E(E_1)$ and $e_2 \in E(E_2)$, three weights $w_1, w_2, w_3 \in \mathbb{R}$, and a partial relation $R \subseteq E(E_1) \times E(E_2)$, the gain function $G : R \times E(E_1) \times E(E_2) \to \mathbb{R}$ is $G(R, e_1, e_2) = -1$ if e_1 and e_2 do not have the same label. Otherwise, $G(R, e_1, e_2) = \sum_{i=1}^{3} w_i \cdot c_i(R, e_1, e_2)$.

In Section VI, we have set the weights to be $w_1 = 3$, $w_2 = 15$, and $w_3 = 1$.

It is important to note that the gain function G is dynamic, i.e. depends on the current relation R.

Example 3.12: Consider the explanations E_1 and E_2 depicted in Figures 1a, 1b, and let the partial relation be $R = \{((paper_3, Carol), (paper_4, Dave))\}$. Then the gain of the pair $((paper_3, \text{Erdős}), (paper_4, \text{Erdős}))$ is $w_1 \cdot 1 + w_2 \cdot 2 + w_3 \cdot 1$.

Algorithm 1 creates a complete relation from the edges of the two explanations E_1, E_2 and converts this relation into a query with minimum variables w.r.t that relation. The input of the algorithm is two explanations E_1, E_2, and a natural number $numIter$ which will be the number of times the algorithm will try to find a complete relation with the maximal gain. Its main loop consists of $numIter$ iterations, where in each iteration, a different partial relation is found. In the i'th iteration a priority queue of all pairs of edges (e_1, e_2) is created (line 4), and ordered by the gain function depicted in Definition 3.11. To create a variety of relations, the top $i - 1$ ranked pairs are removed from the queue (line 5). After that, a relation is created based on the gain of each pair of edges. This is done in an inner loop which iterates as long as the priority queue is non-empty, and there are edges that have not yet been paired. In the first inner iteration, the algorithm pops the highest ranking pair of edges adjacent to the distinguished nodes and adds it to the relation $curRel$ (lines 11–12). During each subsequent iteration, it pops the first pair of edges in the queue, i.e., the pair with the highest gain (line 14). It then adds the chosen pair to the relation (line 15), updates the status of the edges to be paired, updates the variable $curGain$, and recomputes the gain of the remaining pairs in the queue (line 18). In the case of a complete relation, the algorithm compares the gain of the current complete relation and the maximal gain so far and if it is higher, it updates the maximal-gain complete relation (lines 19–20). Finally, in line 25 the algorithm assembles from the complete relation $maxRel$ the consistent simple query which was found to have the minimum variables using the procedure described in the proof of Proposition 3.10 (see [16] for details).

Algorithm 1: FindRelationGreedy

input : Two Explanations E_1, E_2, $numIter \in \mathbb{N}$
output: Simple Consistent Query Q

1 $maxRel \leftarrow \emptyset$;
2 $maxGain \leftarrow 0$;
3 **for** $i = 1$ *to* $numIter$ **do**
4 $\quad Q \leftarrow InitPriroryQueue(E(E_1), E(E_2))$;
5 $\quad FirstElemts \leftarrow$ first $i - 1$ pairs in Q;
6 $\quad Q.remove(FirstElemts)$;
7 $\quad curRel \leftarrow \emptyset$;
8 $\quad curGain \leftarrow 0$;
9 $\quad \forall\, e \in E(E_1) \cup E(E_2)$ $e.paired \leftarrow false$;
10 $\quad$ **while** $Q \neq \emptyset$ *and* $\exists\, e \in E(E_1) \cup E(E_2)$ *s.t.* $e.paired = false$ **do**
11 $\quad\quad$ **if** *Distinguished nodes are not paired yet* **then**
12 $\quad\quad\quad$ $(e_1, e_2) \leftarrow$ First pair in Q where the source/target of E_1, E_2 is the distinguished node;
13 $\quad\quad$ **else**
14 $\quad\quad\quad$ $(e_1, e_2) \leftarrow Q.pop()$;
15 $\quad\quad$ $curRel \leftarrow curRel \cup \{(e_1, e_2)\}$;
16 $\quad\quad$ $curGain \leftarrow curGain + G(curRel, e_1, e_2)$;
17 $\quad\quad$ $e_j.paired \leftarrow true$ for $j = 1, 2$;
18 $\quad\quad$ $Q.recomputeGains(curRel)$;
19 $\quad\quad$ **if** $\forall e \in E(E_1) \cup E(E_2).\ e.paired = true$ *and* $curGain > maxGain$ **then**
20 $\quad\quad\quad$ $maxRel \leftarrow curRel$;
21 $\quad\quad\quad$ $maxGain \leftarrow curGain$;
22 **if** $maxRel = \emptyset$ **then**
23 $\quad$ **return** "No Consistent Query Found";
24 **else**
25 $\quad$ **return** $BuildQuery(maxRel)$;

A basic requirement of the algorithm is to return a consistent simple query if one exists.

Proposition 3.13: Given two explanations as input, if there exists a simple consistent query w.r.t these explanations, Algorithm 1 will find a simple consistent query.

Example 3.14: Reconsider the example-set consisting of the explanations E_1, E_2 in Figures 1a, 1b, and $numIter = 1$ as input to Algorithm 1. The weights for the gain function (Definition 3.11) are set to $w_1 = 3$, $w_2 = 15$, and $w_3 = 1$. $minMatch$ is initialized to $\emptyset$ and the main loop at line 3 start its its single iteration. After computing the gain function for each pair of edges, the pair $(paper_3, \text{Erdős})$ and $(paper_4, \text{Erdős})$, has the highest gain since the two edges have the common target Erdős, so this pair is the first to be chosen for $curRel$ in lines 14–17. Now the gain function is recomputed so the pair $(paper_3, Carol)$ and $(paper_4, Dave)$ has a higher gain (since the sources $(paper_3, paper_4)$ were mapped together before). The gain of the other pairs of the form $((u, v), (paper_4, \text{Erdős}))$ and $((paper_3, \text{Erdős}), (u, v))$ is lower, as one edge in the pair is already paired. Thus, in the second inner iteration, the algorithm adds the pair $(paper_3, Carol)$, $(paper_4, Dave)$ to the set $curRel$. Recomputing the gain function now will give a lower gain to all pairs of the form $((u, v), (paper_4, \text{Erdős}))$, $((paper_3, \text{Erdős}), (u, v))$, $((u, v), (paper_4, Dave))$, $((paper_3, Carol), (u, v))$ (as they are already paired), except for $(paper_3, Carol), (paper_4, \text{Erdős})$ and $(paper_3, \text{Erdős}), (paper_4, Dave)$ which will now have the lowest gain. The pair $(paper_2, Carol)$, $(paper_4, Dave)$ has the highest gain so it is added to $curRel$. Continuing on this computation will result in the complete relation that pairs all edges in E_1 to the edge $(paper_4, Dave)$ except for the edge $(paper_3, \text{Erdős})$ paired with the edge $(paper_4, \text{Erdős})$. $curRel$ satisfies the condition in line 19, so $maxRel$ is updated in line 20. Since this is the highest gain relation, at line 25, the algorithm assembles Q_1 (see Figure 2a) from the set $maxRel$ using procedure $BuildQuery$.

Denote $|E(E_1)| = m_1$ and $|E(E_2)| = m_2$. The complexity of the algorithm is $O(numIter(m_1 \cdot m_2)^2 \log(m_1 \cdot m_2))$.

Naturally (given our hardness result), the algorithm is not guaranteed to find the optimal query. We provide an example of such a case in the full version.

Extending to n Explanations: In the general case, the input may include more than 2 explanations. In this case, we run Algorithm 1 on each pair of explanations and greedily choose to merge the pair of explanations yielding the complete relation with maximal gain. We repeat this procedure while there are explanations to merge in the example-set, merging not only explanations with other explanations but also explanations with intermediate queries. A subtlety here is that we now generalize pairs of graphs which are not necessarily explanations but also intermediate queries. Note that our algorithm already tries to pair edges that *add* the least amount of variables to the set $maxRel$, so when merging two queries, pairing edges with the same constants will cost less (Definition 3.11). Moreover, merging a consistent query w.r.t a set of explanations $\mathcal{E}_1$ with another consistent query w.r.t a different set of explanations $\mathcal{E}_2$ using Algorithm 1, creates a consistent query w.r.t $\mathcal{E}_1 \cup \mathcal{E}_2$. This is a conclusion of Proposition 3.13: Suppose we have merged queries Q_1, Q_2 into the query Q using Algorithm 1, and queries Q_1, Q_2 are, in turn, consistent w.r.t $\mathcal{E}_1, \mathcal{E}_2$, respectively. Then there is a match $\mu_1 : Q \to Q_1$, and there is also a match $\mu : Q_1 \to E$ for every $E \in \mathcal{E}_1$, so $\mu \circ \mu_1 : Q \to E$ is a match (the same argument is valid for Q_2).

IV. Unions of Simple Queries

We have focused so far on inferring only simple queries; in this section we consider an extension of the solution to the class of union queries. Note that the objective function of minimizing the number of variables is not longer adequate when union is allowed: the trivial query that treats each explanation as a simple query and unions all of them has no variables, but is obviously an over-fit. A more suitable solution is to generalize some of the explanations and unify the rest.

To this end, we formally define the *minimum generalization (MG) problem* whose input is an example-set and output a union of simple queries that minimizes a balance between the number of unions and the number of variables.

Definition 4.1: Given an example Ex with k explanations, the solution to the minimum generalization problem is a query Q which is a union of simple SPARQL queries such that (1) for each $1 \leq i \leq k$ there a simple query $q \in Q$ s.t. E_i is in the set $prov_q(dis(E_i))$ (i.e., when running q on E_i we get the result $dis(E_i)$ with the provenance graph E_i), and (2) Given weights $w_1, w_2 \in \mathbb{R}$ the function $f(Q) = w_1 \cdot \sum_{q \in Q} |vars(q)| + w_2 \cdot |Q|$ is minimized among the queries that satisfy (1).

Example 4.2: Consider the example $Ex = \{E_1, E_2\}$, where E_1, E_2 are given in Figures 1a, 1b. Also consider the queries $Q = Union(\{E_1, E_2\})$ (i.e. a query with only constants whose projected nodes are the distinguished nodes in the explanations) and Q_1 depicted in Figure 2a. First, Q clearly satisfies condition (1) and Q_1 also satisfies it since its projected node is $?a_1$ and we have seen in Example 3.3 that it matches both E_1 and E_2 with the output being their distinguished nodes. Computing the cost function f for Q gives us $f(Q) = w_1 \cdot (0+0) + w_2 \cdot 2$ and $f(Q_1) = w_1 \cdot 6 + w_2 \cdot 1$.

The NP-hardness result of the previous section goes through and so we propose another greedy algorithm, leveraging the solution for simple queries presented in the previous section.

Greedy Approach: We now present a greedy algorithm for inferring a union of simple SPARQL queries. In each step of Algorithm 2 we merge two simple queries whose merging cost is the smallest. We continue this process as long as there is a merge that decreases the cost function. Specifically, in lines 1, 2 we initialize the query Q to be a union query where each simple query contains only constants as we have exemplified before. In each iteration of the loop in line 4, we update the previous cost to the current one (line 5), call procedure $MergeBestTwo$ (line 6) to update the query Q, and update the current cost function (line 7). When the cost function can no longer be decreased by following this process we return

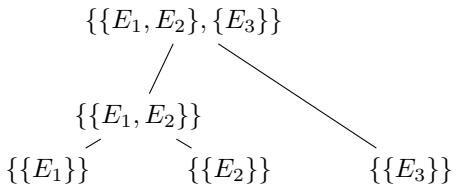
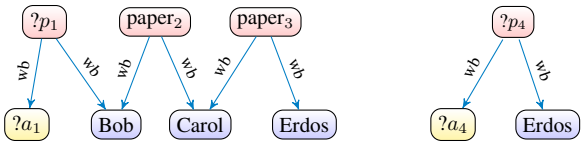


Fig. 3: Part of the Lattice in Example 4.3



(a) Q_3 (b) Q_4

Fig. 4: Union Queries

the query. The $MergeBestTwo$ procedure runs Algorithm 1 on all pairs of simple queries in the current Q in search of the pair of simple queries whose consistent simple query has the minimum number of variables.

Algorithm 2: FindConsistentUnion

input : Example-set Ex, weights w_1, w_2
output: A consistent SPARQL union of simple queries

1 $Q \leftarrow Union(Ex)$;
2 $costCur = f(Q)$;
3 $costPrev = \infty$;
4 **while** $costCur < costPrev$ **do**
5 $\quad costPrev \leftarrow costCur$;
6 $\quad Q \leftarrow MergeBestTwo(Q)$;
7 $\quad costCur \leftarrow f(Q)$;
8 **return** Q;

Abstractly, Algorithm 2 traverses a lattice whose elements are sets of sets of explanations. The leaves are sets, containing a set with a single explanation from the example-set.

Example 4.3: Consider the example-set $Ex = \{E_1, E_2, E_3\}$, where the explanations are depicted in Figure 1. As input, we take Ex and $w_1 = 2$, $w_2 = 5$. The algorithm starts by setting $Q = Union(\{E_1, E_2, E_3\})$ and computing $costCur = 15$ (we have no variables and three simple queries in the union). It then starts the first iteration of the while loop (line 4): setting $costPrev$ to be 15 (line 5), using the $MergeBestTwo$ procedure to merge the two explanations whose merge will most decrease the cost, which are E_1 and E_3, to get the simple query Q_3 depicted in Figure 4a (line 6), and ending the iteration by computing $costCur = 14$ (two variables and two simple queries in the union) in line 7. At the end of the iteration, $Q = Union(\{Q_3, E_2\})$. In the second iteration, the algorithm again updates $costPrev$ to be 14 and tries to merge Q_3 and E_3. This merge yields the simple query Q_1 in Figure 2a. By computing $costCur = 2 \cdot 6 + 5 = 17$, the algorithm verifies that the resulted query does not yield a lower cost, so the algorithm breaks from the loop with $Q = Union(\{Q_3, E_3\})$ which is returned in line 8.

Top-K Queries: As both our cost function and the algorithms for optimizing it are heuristic, it is worthwhile to retrieve multiple candidates rather than focus on a single query. The candidates may then be filtered based on user feedback, as discussed in the next section.

Algorithm 2 can be adapted quite naturally to output k queries. We augment procedure $MergeBestTwo$ in Algorithm 2. Instead of choosing the best pair of simple queries to merge in each iteration, in the first iteration, the procedure chooses the top-k best pairs which yield the simple queries with minimal cost. The output of the procedure is now a list of top-k simple queries, thus outputting k different sets. In every subsequent iteration, the procedure tries to generalize the best pair from every set, ending up with k^2 sets, and choosing the top-k sets that minimize the cost function. Note that this is again only a heuristic solution: as we filter the top-k queries in each step, we are not guaranteed to find the top-k queries overall (recall the NP-hardness of the case $k = 1$).

Example 4.4: Reconsider as input to the augmented Algorithm 2 the example-set composed of the four explanations in Figure 1 and the weights $w_1 = 1$, $w_2 = 7$. We demonstrate the computation of the top-3 queries.

We begin from the four separate explanations giving the value 28 to the cost function. In the first iteration, the algorithm suggests the three pairs of merges that lead to a minimized cost function. In our case, the three pairs are (E_1, E_3), (E_2, E_4), (E_2, E_3) resulting in the costs 23, 23, 27, respectively. Thus far, we have the potential queries $Union(\{Q_3, E_2, E_4\})$, where Q_3 is depicted in Figure 4a, $Union(\{Q_4, E_1, E_3\})$, where Q_4 is depicted in Figure 4b, and $Union(\{Q_1, E_1, E_4\})$, where Q_1 is depicted in Figure 2a. The algorithm continues by examining possible merges in each of these cases. First, note that the query including Q_1 is already consistent with the entire example-set so, the possible query $Union(\{Q_1, E_1, E_4\})$ results in Q_1 at the cost of 13. As for the possible queries $Union(\{Q_3, E_2, E_4\})$, the algorithm suggest the merge of E_2, E_4 resulting in the query $Union(\{Q_3, Q_4\})$ with the cost of 18. Similarly, for the possible query $Union(\{Q_4, E_1, E_3\})$ the algorithm suggests the merge of E_1 and E_3 again resulting in the query $Union(\{Q_3, Q_4\})$. The algorithm also remains with the query $Union(\{Q_4, E_1, E_3\})$ costing 23. Therefore, the top-3 queries in this case are Q_1 (costing 13), $Union(\{Q_3, Q_4\})$ (costing 18), and $Union(\{Q_4, E_1, E_3\})$ (costing 23).

V. Feedback

As outlined in the Introduction, we propose an interactive approach for inferring the user-intended query. An important feature is the use of disequality constraints, both as a feedback tool for distinguishing between different queries, and also apply them in the query itself.

Disequalities: So far we have focused on inferring queries without disequalities between variables. For each of the top ranked queries returned by our algorithm, we may add disequalities between every pair of variables that are mapped

to nodes of the same type (this is an additional information in the ontology) but with different values in all explanations.

Example 5.1: Reconsider the example-set Ex composed of the explanations E_1, E_2, E_3, E_4 depicted in Figure 1, and the consistent query Q_1 in Figure 2a, the algorithm now considers adding disequalities to Q_1. It Starts by examining the values assigned to the variables $?a_1, ?a_2$ when matched with each of E_1, E_2, E_3, E_4. These are respectively $L(?a_1) = [Alice, Dave, Felix, Harry]$ and $L(?a_2) = [Bob, Dave, Bob, Harry]$. Since Dave was assigned to both for a single explanation (E_2) we may not add an disequality. This will be the case for all pairs of nodes of the same type in Q_1 so no disequalities may be introduced. Consequently, the results of Q_1 will also include authors with Erdos number 2. In contrast, recall query $Q = Union(\{Q_3, Q_4\})$ from Example 4.4 (Q_3, Q_4 are depicted in Figure 4), and consider the pair of nodes $?a_1, Bob$. $L(?a_1) = [Alice, Felix]$ and the disequality $?a_1 \neq Bob$ is thus possible. The disequalities $?a_1 \neq Carol$, $?a_1 \neq Erdos$, $?p_1 \neq paper_2$, $?p_1 \neq paper_3$, and $?a_4 \neq Erdos$ are similarly possible.

Choosing one query: We now have a set of candidate queries: the top-k queries obtained in the previous section along with every possible choice of disequalities between relevant variables. We wish to focus on a single query that matches the user intention, by asking for user feedback. There are two main challenges here: how to present questions, and how to reduce their number. We start with the former.

Feedback Using Provenance: To choose between two queries, there are multiple ways to ask for feedback, assuming that the user is unable to understand formal queries. One possible solution is to asks the user whether a specific node, which is in the output of one candidate query but not in that of the other one, should be included in the result of their intended query or not. It may however be non-trivial to answer such question:

Example 5.2: Reconsider queries Q_1 from Figure 2a and $Q = Union(\{Q_3, Q_4\})$ described in Figure 4. Showing the user a result of Q_1 which is not a result of Q, could mean to show an author with an Erdős number exactly 2, or an author associated with Erdős through a path not including Bob and Carol – but to answer whether such an author should appear in the result set is nontrivial.

To this end, we turn to provenance again. *We do not simply display the result but also include its provenance, with respect to the candidate query of which it is a result.* Provenance is displayed to the user as a graph showing the relations of the output example to the other relevant nodes (more on this in Section VI-A).

Example 5.3: Reconsider the previous example. The information presented to the user will not just include the author but also its provenance which explicitly shows the length-2 co-authorship path to Erdős, thus clearly explaining the rationale of returning this answer and allowing the user to verify whether this logic is relevant for her (i.e. to understand that answering "yes" will include in the query result all authors with Erdős number 2).

Difference Queries: A näive solution is to run a pair of queries Q_i, Q_j, while storing the result set and provenance of the two queries. This may in general be costly in time and space. Instead, when considering a pair of queries Q_i, Q_j we run the difference query $Q_i - Q_j$ without provenance tracking; then to obtain the provenance of a certain result, we bind this result tuple to Q_i and compute its provenance.

Example 5.4: Reconsider the queries Q_1 depicted in Figure 2a and $Union(\{Q_3, Q_4\})$ depicted in Figure 4. To distinguish between them, we run e.g. $Q_1 - Union(\{Q_3, Q_4\})$ to only get some authors with Erdős number 3 (those not connected through Bob and Carol with $paper_2$ and $paper_3$) and authors with Erdős number 2.

We want to ensure that users do not disqualify a query because of extra disequalities placed in the query. Our solution is based on the observation that every additional disequality added to the query can only reduce its the number of results. Consequently, when asking users to distinguish between two queries Q_i, Q_j, we take the difference query $Q_i^{\text{all}\neq} - Q_j^{\text{no}\neq}$, where the former is Q_i with all possible disequalities (as demonstrated in Example 5.1) and the latter is Q_j with no disequalities at all. This way, if the user answers that a result is relevant, we can disqualify all "forms" of Q_j (with any subset of the diseqalities). If the result is not relevant, we can similarly disqualify all forms of Q_i. The result of the difference query may be empty in which case we simply consider a different pair of queries.

Algorithm 3: User Feedback

input : A set $\mathcal{Q} = \{Q_1, \ldots, Q_k\}$ of k SPARQL queries
output: Single query Q

1 **while** $|\mathcal{Q}| > 1$ **do**
2 $\quad Q_i^{\text{all}\neq}, Q_j^{\text{no}\neq} \leftarrow ChooseArbitraryPair(\mathcal{Q})$;
3 $\quad Q_{diff} \leftarrow Q_i^{\text{all}\neq} - Q_j^{\text{no}\neq}$;
4 $\quad DiffResults \leftarrow Eval(Q_{diff})$;
5 $\quad$ **if** $DiffResults \neq \varnothing$ **then**
6 $\quad\quad res \leftarrow SampleRand(DiffResults)$;
7 $\quad\quad Q_b \leftarrow bind(Q_i^{\text{all}\neq}, res)$;
8 $\quad\quad resProv \leftarrow Eval(Q_b)$;
9 $\quad\quad answer \leftarrow AskUser(resProv)$;
10 $\quad\quad$ **if** *answer is Yes* **then**
11 $\quad\quad\quad \mathcal{Q} \leftarrow \mathcal{Q} \setminus \{Q_j\}$;
12 $\quad\quad$ **else**
13 $\quad\quad\quad \mathcal{Q} \leftarrow \mathcal{Q} \setminus \{Q_i\}$;
14 **return** $\mathcal{Q}$;

Algorithm 3 iteratively eliminates the queries by choosing an arbitrary pair from the set $\mathcal{Q}$, Q_i, Q_j (line 2), generating the difference query of the pair as explained above (line 3), and evaluating it (line 4). If the result set is not empty, the algorithm samples one result (line 6) and then binds the result to Q_i, and evaluates the binded query (lines 7, 8). It then asks the user if the sample result and its provenance should be included in the desired results (line 9). A "yes" ("no") answer

means Q_j (respectively Q_i) is removed from $\mathcal{Q}$ (lines 10, 11).

Example 5.5: We now exemplify the operation of Algorithm 3 using the top-3 queries in Example 4.4 as input (the queries already include disequalities). We enter the while loop (line 1) in the algorithm with $\mathcal{Q} = \{Q_1, Union(\{Q_3, Q_4\}), Union(\{Q_4, E_1, E_3\})\}$, and take the pair Q_1, $Union(\{Q_4, E_1, E_3\})$ in line 2. Therefore, we evaluate $Q_{diff} = Q_1 - Union(\{Q_4, E_1, E_3\})$, where Q_1 contains all possible disequalities listed in Example 5.1 and $Union(\{Q_4, E_1, E_3\})$ contains none (lines 3, 4). One of the results of Q_{diff} is the author Felix whose Erdős number is 3, but through an authorship path that does not include Bob and Carol (thus the condition in line 5 holds), so we bind this value to Q_1 and evaluate this query (lines 7, 8) resulting in the provenance for Felix (similar to the explanations in Figures 1a, 1c) stating that its Erdős number is 3. When this result is shown to the user, she responds "yes" (lines 9, 10), i.e. this is a desirable result and explanation, so the query $Union(\{Q_4, E_1, E_3\})$ is discarded from $\mathcal{Q}$ in line 11. Next, the only pair remaining is $Q = Union(\{Q_3, Q_4\})$ and Q_1, thus the algorithm assigns $Q_{diff} = Q - Q_1$ (where, again, Q contains all disequalities and Q_1 contains none), and evaluates it. Since the result of Q_{diff} is empty, we continue to the next iteration where the algorithm again chooses the same pair but this time assigns $Q_{diff} = Q_1 - Q$ and evaluates it. The result set contains William whose Erdős number is 2 through an authorship path that does not include a joint paper with Bob. When this result is shown to the user, she responds "no", Q_1 is discarded from $\mathcal{Q}$ and we are left with $\mathcal{Q} = \{Q\}$, which is returned to the user. Another possible result of the difference query is an author whose Erdős number is 3 through a different path than the one including Bob, Alice, $paper_2$, and $paper_3$. The provenance will be presented to the user and she will again respond "no", resulting in the same outcome.

Once we have focused on a query pattern, choosing a single query $Q \in \mathcal{Q}$, Q can be returned automatically with the maximal number of disequalities, but users may want a less restrictive query. Thus, we can use a slight augmentation of Algorithm 3, but this time for choosing the right disequalities. Assume there are d possible disequalities in Q. We begin by assigning to Q_j the query Q with all d disequalities and assigning to Q_i the query Q with $d-1$ disequalities (removing an arbitrary disequality). If the user is interested in including the difference result $Q_i - Q_j$, we repeat this stage with Q_i as Q_j. If the user is not interested in the difference result or $Q_i - Q_j$ does not yield any result, we try to remove a different arbitrary disequality. Finally, if all difference results are empty, we start assigning to Q_i the query Q_j without *two* arbitrary disequalities and repeat the process described above (if this reoccurs, we remove three disequalities and so on). An optimization for this process is to store each disequality the user has approved (in the case where she is not interested in the result of $Q_i - Q_j$) and never ask about it again (it will be included in the final query). If the difference result is never empty, the process terminates after d steps since we either disqualify or approve a disequality in each step.

VI. Implementation and Experiments

A. System Architecture

Figure 5 depicts the architecture of our system, named `QuestPro` [18]. The system is implemented in JAVA 8 and uses the ARQ Jena package [19] to evaluate queries and track provenance. Its web UI is built using JavaScript packages including react, d3 and material UI, and runs on Ubuntu Xenial Xerus. Sections III and IV have focused on the "Query by Provenance" component, and Section 5 has focused on the feedback loop described in the right part of the architecture. For the formulation of the example-set, the "Ontology visualizer" component helps users in the formulation of explanations and compiles them into provenance. In the first stage, the user provides output examples. To formulate the explanation for this output example, the GUI first displays the 1-neighborhood of the selected output node in the ontology. At this stage, the user can browse the nodes graphically and select which node in the neighborhood best describes the *reason* for his output example. Because there can be many nodes in this subgraph, users can filter edges by their labels, leaving only nodes connected through relevant labels to the output node. If users wish to continue to form a larger graph, with more than two nodes, they can continue to see the 2-neighborhood of the chosen node and so on. Note that the users may form a subgraph that is not necessarily a path. The GUI assists with user interaction in the two parts of the system: showing the neighboring nodes of the output example to help users formulate explanations, and visualizing the provenance in a graph form in order to assist users in selecting the correct query during the feedback stage (Algorithm 3).

B. Automatic Experiments

We have performed experiments with queries from the benchmarks SP2B [20] (a DBLP-based ontology containing information about authors, conferences, and papers), and BSBM [21] (an e-commerce ontology, containing products, vendors, and consumer reviews about the products). We have ran the experiments on fragments of DBpedia, SP2B, and BSBM ontologies sized 42MB, 67MB, and , 647.5MB, respectively. The main significance of the size of the fragment is to allow for enough variety for sampled output examples and explanations. The queries have 1-12 edges, 1–12 variables, and multiple instances of joins [21], [20]. We have considered queries 1v0–10v0 from BSBM and queries 2, 3a, 3b, 6, 8a, 8b, 11, 12a from SP2B as a representative sample (queries 4v0 ,7v0, 9v0 from BSBM and queries 4,7 from SP2B were not included in the experiments since they are designed to output a single result even when evaluated on the entire ontology while we require at least two explanations to reproduce a query). To examine our approach synthetically, we first evaluated the queries using the provenance-aware query engine described in Section VI-A, and then for each query, used a random sample of the results and their provenance as input to our algorithm for it to reverse engineer the query while measuring runtime. The choice of examples matters a lot, and thus we repeat each

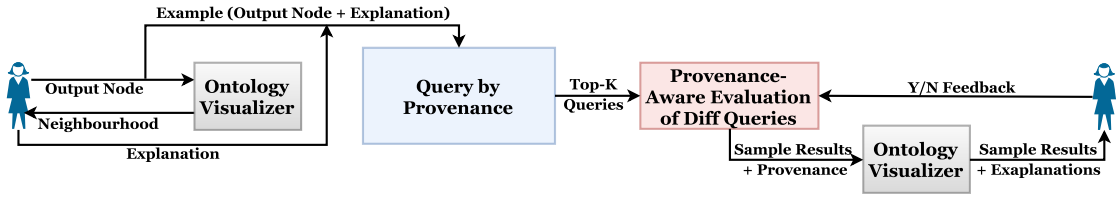
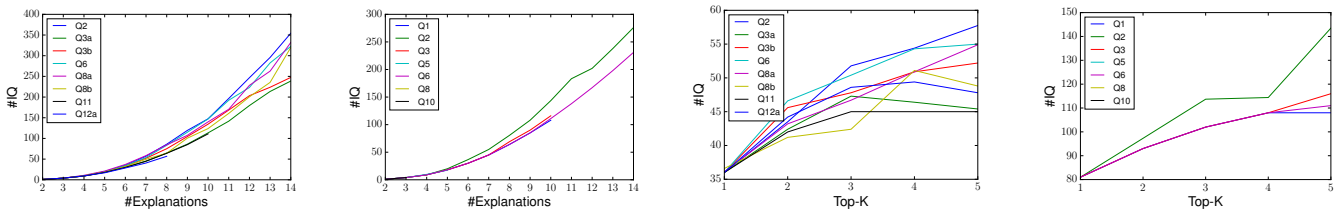
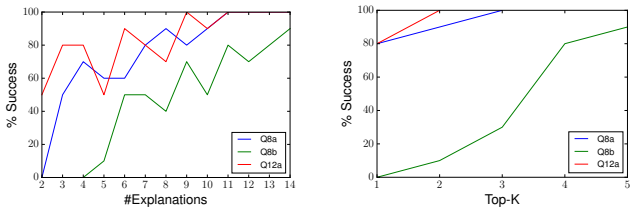


Fig. 5: System Architecture



(a) SP2B (b) BSBM (c) SP2B (d) BSBM

Fig. 6: Number of Calls to Algo. 1 By Algo. 2 as a Function of the Number of Explanations and k



(a) Increasing Exp. (b) Increasing k

Fig. 7: % of Success in Reverse-Engineering SP2B Queries

experiment 10 times to reduce the effect of the random choice. All experiments were performed on a i7 processor and 16GB RAM with Ubuntu Xenial Xerus (16.04).

Quality: To test the quality of `QuestPro`, we have examined what is the minimum number of explanations needed for each of the BSBM and SP2B queries to be placed in the k candidate queries returned by Algorithm 2. The rationale is that once the original query is part of this set, the user feedback can allow the system to focus on it. We report the success rate of the algorithm for the 10 samples in each point. Our solution was very successful for the BSBM queries – for all of the above-mentioned queries, which are all inspected queries that fall in our class (union of simple queries with disequalities), the original query was ranked first in the top-k candidates already for 2 explanations. For the SP2B ontology the results are mixed: queries q2 (9 edges), q3a (2 edges), 3qb (2 edges) and q11 (1 edge) were found in a 100% of the runs with only 2 explanations. For query q6 (5 edges), its pattern was found but contained an extra constant in all examples since all of its provenance examples in the SP2B ontology contain this constant (due to this fact it was not included in the graph). Queries q8a (6 edges, authors with Erdős numbers 1,2,3) and q12a (6 edges) reached a success rate of 100% with 11 explanations. Query q8b (authors with Erdős numbers 1 or 3) that included 8 edges was inferred in 90% of the runs with 14 explanations. For the three latter queries we show in Figure 7 the results as a function of the number of explanations (fixed $k = 4$) and k (fixed number of explanations to be 11). Unsurprisingly, the quality generally improves as either the number of explanations or k grows, with variability that is due to the effect of random choices of explanations.

Execution Times: We now detail the execution times of our top-k algorithm for an increasing number of explanations and a fixed $k = 3$, when trying to reverse-engineer the queries of SP2B and BSBM. The execution time, for 7 explanations, was generally less than 0.5 seconds for most queries, with the only exceptions being SP2B query q12a (1.34 seconds) and BSBM query q2v0 (5.8 seconds).

Number of Intermediate Queries: Finally, we have examined the effect of the number of explanations and the parameter k on the number of intermediate queries considered in Algorithm 2. Namely, this is the number of times Algorithm 2 calls Algorithm 1 in line 6 during its run. Figures 6a and 6b show the number of intermediate queries as a function of the number of explanations for the SP2B and BSBM ontologies, respectively. We observe that as the number of explanations increases (fixing $k = 5$), the number of intermediate queries that has to be considered in each run grows up to a point of more than 260 matches examined for 14 explanations in query 2v0. Indeed, having more explanations in the example-set gives rise to more options of merges and unions that are considered by the algorithm. Figures 6c and 6d show the number of intermediate queries as a function of k (fixing the number of explanations to 7 for SP2B and 10 for BSBM). Here also the number of intermediate queries that has to be considered grows larger (as a general trend, with some exceptions that are due to the random choices of examples), but this time more moderately.

C. Experience With `QuestPro`

We have examined the usability of our approach through interaction with 9 users who are proficient in writing SPARQL queries; our goal was to assess the usability of our approach. We have used the DBpedia ontology [22] and queries related to movies. We have shown each user the textual description of the 10 queries in Table I, and asked them to choose two queries out of queries 1–5 (basic queries) and two queries out

TABLE I: User Study Queries

Query	Query Description
1	`All actors who co-starred with Brad Pitt`
2	`All actors who starred in Titanic`
3	`All movies starring Kate Winslet`
4	`All actors born in the UK`
5	`All actors who starred in a Tim Burton movie`
6	`All actors who starred in a Quentin Tarantino movie with Samuel L. Jackson`
7	`All actors who also are also directors`
8	`All actors who starred in a movie with an actor who co-starred in a movie`
9	`All actors who starred in a Quentin Tarantino and are also married to an actor`
10	`All movies that were written and directed by the same person`

of queries 6–10 (more challenging queries), and for each one that they chose, formulate examples and explanations through the system so that the inferred query will have the same semantics as the query described in the corresponding row of Table I. Figure 8 shows the number of successful interactions (blue), failed interactions (red) and redone interactions that were successful after redo (green). Each interaction is an attempt at the formulation of some query. For these queries, the system was mostly successful at the inference of the queries based on the given explanations, except for six specific cases, as follows. One user who attempted to provide explanations for query 3 had to start over because he did not fully understand the UI. In one interaction regarding query 4, the user gave an explanation that included London and the system added it to the inferred query using union. In one attempt of inferring query 6, the user had to redo the task since he forgot to input an explanation and got the wrong query, and in the other the user did not fully understand the query, so he made a mistake and had to start over. Regarding query 9, one user did not give a complete explanation and thus obtained a query which unioned the two explanations provided, and a different user confused the direction of the arrows in the ontology and selected a different relation than intended.

We have also asked the users to freely formulate an example of their own, related to DBpedia. 6 users have succeeded in providing examples and explanations for a query of their own and got the intended query. 2 other users got the query they were aiming for except for incorrect disequalities. In these cases, the system has inferred the right graph structure for the query, but was unable to deduce the correct disequalities from the explanations given. Another user has obtained the intended query with an extra constant since all explanations she chose contained that constant. Namely, she tried to get the query returning all actors who played in more than one Tarantino movie but provided two explanations with identical parts, so the inferred query was more specific than the intended one.

Summary: In total, our automatic experimental study included 15 queries. In all cases except one (q8b), the query was inferred using at most 11 explanations (in fact, 11 of the 15 were found with only 2 explanations), showing the effectiveness of our approach. Our experience with `QuestPro` shows that, while formulating explanations is a non-trivial task, users are mostly successful at interacting with the system (30 successful attempts and 2 successful redos out of 36), and the system in turn is able to infer the correct query through the interaction. Our initial user experience indicates the usefulness of our system. An extensive, full-fledged, user study is left for future work. In particular, we plan to compare our approach with an approach that is based only on output examples and with directly writing queries.

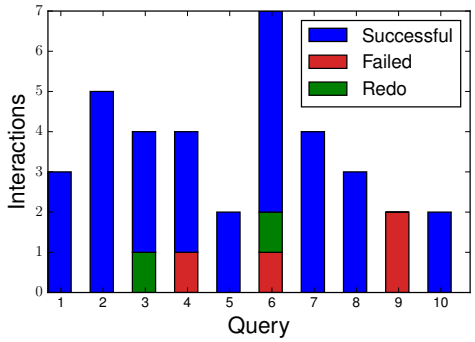


Fig. 8: Attempts in User Study Queries

VII. Related Work

There is a large body of literature on learning queries from examples, in different variants. A first axis of these variants concerns learning a query whose output *precisely* matches the example (e.g. [10], [11]), versus one whose output contains the example tuples and possibly more (e.g. [12], [13], [14] and the somewhat different problem in [23]). The first is mostly useful e.g. in a use-case where an actual query was run and its result, but not the query itself, is available. The second is geared towards examples provided manually by a user, who may not be expected to provide a full account of the output. Another distinguishing factor between works in this area is the domain of queries that are inferred; due to the complexity of the problem, it is typical (e.g. [11], [14], [15]) to restrict the attention to join queries, and many works also impose further restrictions on the join graph [10], [24]. We do not impose such restrictions and allow for union and and disequalities in addition to join. There is also a prominent line of work on query-by-example in the context of *data exploration* [25], [15], [26], [27], [28]. Here users typically provide an initial set of examples (either only positive, or positive and negative), leading to the generation of a consistent query (or multiple such queries); the queries and/or their results are presented to users, who may in turn provide feedback used to refine the queries, and so on. Automatic inference of queries has also been studied in the context of SPARQL [29], [30].

The fundamental difference between our work and previous work in this area is the assumed input. Our work is the first, to our knowledge (with the exception of our short demo paper [31] and an unpublished technical report focused on semiring provenance and the relational settings [32]), that bases its inference of queries on explanations that form provenance information. Leveraging this additional information, we are

able to reach a satisfactory query (1) for a relatively large class of queries, (2) with a small number of examples, (3) with no assumption or information on the underlying schema. Our UI also has an intuitive and graphical presentation of the provenance that assists users in formulating explanations and giving feedback in the final stage of choosing the query.

Data Provenance has been extensively studied (see e.g. [4], [1], [33], [34], [5], [35], [36], [37]). Many different models have been proposed, and shown useful in a variety of applications, including program slicing [38], factorized representation of query results [37], and "how-to" analysis [39], [6]. Works focusing on provenance for SPARQL [8], [9], [40], [41] rely on similar principles, proposing algebraic structures to capture operators included in SPARQL and adapting [42] to track provenance for SPARQL updates. We have focused on learning queries from explanations based on graph provenance for SPARQL queries rather than the relational model.

VIII. Conclusion

We have formalized and studied in this paper the problem of inferring SPARQL queries from output examples and their explanations. We have proposed a generic model, based on a simple provenance model. We have further presented solutions that infer simple queries and their union from explanations, extended them to account for disequalities and for proposing multiple options, prompting user feedback to direct the search. We again use provenance for feedback, as means to explain the candidate queries to the users. We have theoretically analyzed and experimentally demonstrated the effectiveness of the approach in inferring queries.

We see our main contribution in the conceptual framework that, for the first time, leverages provenance for a query-by-example solution. Our results may be seen as an indication for the feasibility of such framework. Intriguing directions for future study include a theoretical analysis of the quality of our heuristic algorithms; supporting further expressive query languages and additional operators of the inferred queries, such as OPTIONAL; designing further means of assisting users in formulating explanations and providing feedback; and dealing with incorrect provenance provided by users.

Acknowledgments: This research has been partially funded by the ISF (978/17, 1636/13) and the Blavatnik Interdisciplinary Cyber Research Center (TAU ICRC). The contribution of Amir Gilad is part of a Ph.D. thesis research conducted at Tel Aviv University.

References

[1] T. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *PODS*, 2007.
[2] T. J. Green, "Containment of conjunctive queries on annotated relations," in *ICDT*, 2009.
[3] P. Buneman, S. Khanna, and W. chiew Tan, "Why and where: A characterization of data provenance," in *ICDT*, 2001.
[4] A. D. Sarma, M. Theobald, and J. Widom, "Exploiting lineage for confidence computation in uncertain and probabilistic databases," in *ICDE*, 2008.
[5] J. Cheney, L. Chiticariu, and W. C. Tan, "Provenance in databases: Why, how, and where," *Foundations and Trends in Databases*, 2009.
[6] N. Bidoit, M. Herschel, and K. Tzompanaki, "Query-based why-not provenance with nedexplain," in *EDBT*, 2014.
[7] D. Deutch, A. Gilad, and Y. Moskovitch, "Selective provenance for datalog programs using top-k queries," *PVLDB*, 2015.
[8] F. Geerts, T. Unger, G. Karvounarakis, I. Fundulaki, and V. Christophides, "Algebraic structures for capturing the provenance of SPARQL queries," *J. ACM*, vol. 63, no. 1, 2016.
[9] H. Halpin and J. Cheney, "Dynamic provenance for SPARQL updates," in *ISWC*, 2014.
[10] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy, "Query by output," in *SIGMOD*, 2009.
[11] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava, "Reverse engineering complex join queries," in *SIGMOD*, 2014.
[12] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik, "Discovering queries based on example tuples," in *SIGMOD*, 2014.
[13] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas, "Exemplar queries: Give me an example of what you need," in *VLDB*, 2014.
[14] F. Psallidas, B. Ding, K. Chakrabarti, and S. Chaudhuri, "S4: Top-k spreadsheet-style search for query discovery," ser. SIGMOD, 2015.
[15] A. Bonifati, R. Ciucanu, and S. Staworko, "Interactive inference of join queries," in *EDBT*, 2014.
[16] http://www.cs.tau.ac.il/~amirgilad/full/QuestProFull.pdf.
[17] N. Vicas, "Computational complexity of graph compaction," in *PhD Thesis*, 1997.
[18] G. Repository, https://github.com/efratoio/SQBE.
[19] T. A. S. Foundation, "Arq - a sparql processor for jena," https://jena.apache.org/documentation/query/.
[20] M. Schmidt, T. Hornung, M. Meier, C. Pinkel, and G. Lausen, "SP^{2}bench: A SPARQL performance benchmark," in *Semantic Web Information Management - A Model-Based Perspective*, 2009.
[21] B. S. Benchmark, http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/.
[22] DBPedia, http://wiki.dbpedia.org/.
[23] M. M. Zloof, "Query by example," in *AFIPS NCC*, 1975.
[24] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom, "Synthesizing view definitions from data," ser. ICDT, 2010.
[25] T. Sellam and M. L. Kersten, "Meet charles, big data query advisor," ser. CIDR, 2013.
[26] A. Abouzied, J. M. Hellerstein, and A. Silberschatz, "Playful query specification with dataplay," *Proc. VLDB Endow.*, 2012.
[27] A. Bonifati, R. Ciucanu, A. Lemay, and S. Staworko, "A paradigm for learning queries on big data," ser. Data4U, 2014.
[28] K. Dimitriadou, O. Papaemmanouil, and Y. Diao, "Explore-by-example: An automatic query steering framework for interactive data exploration," in *SIGMOD*, 2014.
[29] G. I. Diaz, M. Arenas, and M. Benedikt, "Sparqlbye: Querying RDF data by example," *PVLDB*, vol. 9, no. 13, 2016.
[30] M. Arenas, G. I. Diaz, and E. V. Kostylev, "Reverse engineering SPARQL queries," in *WWW*, 2016.
[31] D. Deutch and A. Gilad, "Qplain: Query by explanation (demo)," in *ICDE*, 2016.
[32] "Full version," http://www.cs.tau.ac.il/~danielde/QueryByProvFull.pdf.
[33] F. Geerts and A. Poggi, "On database query languages for k-relations," *J. Applied Logic*, 2010.
[34] B. Glavic, J. Siddique, P. Andritsos, and R. J. Miller, "Provenance for data mining," in *TaPP*, 2013.
[35] "Prov-overview, w3c working group note," 2013, http://www.w3.org/TR/prov-overview/.
[36] P. Buneman, J. Cheney, and S. Vansummeren, "On the expressiveness of implicit provenance in query and update languages," *ACM Trans. Database Syst.*, 2008.
[37] R. Fink, L. Han, and D. Olteanu, "Aggregation in probabilistic databases via knowledge compilation," *PVLDB*, 2012.
[38] U. A. Acar, A. Ahmed, J. Cheney, and R. Perera, "A core calculus for provenance," *Journal of Computer Security*, 2013.
[39] A. Meliou and D. Suciu, "Tiresias: the database oracle for how-to queries," in *SIGMOD*, 2012.
[40] G. Karvounarakis, I. Fundulaki, and V. Christophides, "Provenance for linked data," in *In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman*, 2013.
[41] Y. Theoharis, I. Fundulaki, G. Karvounarakis, and V. Christophides, "On provenance of queries on semantic web data," *IEEE Internet Computing*, 2011.
[42] P. Buneman, A. Chapman, and J. Cheney, "Provenance management in curated databases," in *SIGMOD*, 2006.