Efficiently Computing Provenance Graphs for Queries with Negation

Seokki Lee, Sven Köhler, Bertram Ludäscher, Boris Glavic

IIT DB Group Technical Report
IIT/CS-DB-2016-03

2016-10

http://www.cs.iit.edu/~dbgroup/

LIMITED DISTRIBUTION NOTICE: The research presented in this report may be submitted as a whole or in parts for publication and will probably be copyrighted if accepted for publication. It has been issued as a Technical Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IIT-DB prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. payment of royalties).
Efficiently Computing Provenance Graphs for Queries with Negation

Seokki Lee∗ Sven Köhler† Bertram Ludäscher‡ Boris Glavic∗
∗Illinois Institute of Technology. {slee195@hawk.iit.edu,bglavic@iit.edu}
†University of Illinois at Urbana-Champaign. {ludaesch@illinois.edu}
‡University of California at Davis. {svkoehler@ucdavis.edu}

Abstract—Explaining why an answer is in the result of a query or why it is missing from the result is important for many applications including auditing, debugging data and queries, and answering hypothetical questions about data. Both types of questions, i.e., why and why-not provenance, have been studied extensively. In this work, we present the first practical approach for answering such questions for queries with negation (first-order queries). Our approach is based on a rewriting of Datalog rules (called firing rules) that captures successful rule derivations within the context of a Datalog query. We extend this rewriting to support negation and to capture failed derivations that explain missing answers. Given a (why or why-not) provenance question, we compute an explanation, i.e., the part of the provenance that is relevant to answer the question. We introduce optimizations that prune parts of a provenance graph early on if we can determine that they will not be part of the explanation for a given question. We present an implementation that runs on top of a relational database using SQL to compute explanations. Our experiments demonstrate that our approach scales to large instances and significantly outperforms an earlier approach which instantiates the full provenance to compute explanations.

I. INTRODUCTION

Provenance for relational queries records how results of a query depend on the query’s inputs. This type of information can be used to explain why (and how) a result is derived by a query over a given database. Recently, approaches have been developed that use provenance-like techniques to explain why a tuple (or a set of tuples described declaratively by a pattern) is missing from the query result. However, the two problems of computing provenance and explaining missing answers have been treated mostly in isolation. A notable exception is [22] which computes causes for answers and non-answers. However, the approach requires the user to specify which missing inputs to consider as causes for a missing output. Capturing provenance for a query with negation necessitates the unification of why and why-not provenance, because to explain a result of the query we have to describe how existing and missing intermediate results (via positive and negative subqueries, respectively) lead to the creation of the result. This has also been recognized by Köhler et al. [20]: asking why a tuple t is absent from the result of a query Q is equivalent to asking why t is present in ¬Q. Thus, a provenance model that supports queries with negation naturally supports why-not provenance. In this paper, we present a framework that answers why and why-not questions for queries with negation. To this end, we introduce a graph model for provenance of first-order (FO) queries (i.e., non-recursive Datalog with negation) and an efficient method for explaining a (missing) answer using SQL. Our approach is based on the observation that typically only a part of provenance, which we call explanation in this work, is actually relevant for answering the user’s provenance question about the existence or absence of a result.

Example 1. Consider the relation Train in Fig. 1 that stores train connections in the US. The Datalog rule r1 in Fig. 1 computes which cities can be reached with exactly one transfer, but not directly. We use the following abbreviations in provenance graphs: T = Train; n = New York; s = Seattle; w = Washington DC and c = Chicago. Given the result of this query, the user may be interested to know why he/she is able to reach Seattle from New York (WHY Q(n,s)) with one intermediate stop but not directly or why it is not possible to reach New York from Seattle in the same fashion (WHYNOT Q(s,n)).

An explanation for either type of question should justify the existence (absence) of a result as the success (failure) to derive the result through the rules of the query. Furthermore, it should explain how the existence (absence) of tuples in the database caused the derivation to succeed (fail). Provenance graphs providing this type of justification for WHY Q(n,s) and WHYNOT Q(s,n) are shown in Fig. 2 and Fig. 3, respectively. There are three types of graph nodes: rule nodes (boxes labeled with a rule identifier and the constant arguments of a rule derivation), goal nodes (rounded boxes labeled with a rule identifier and the goal’s position in the rule’s body), and tuple nodes (ovals). In these provenance graphs, nodes are either colored as green (successful/existing) or red (failed/missing).

Example 2. Consider the explanation (provenance graph in Fig. 2) for question WHY Q(n,s). Seattle can be reached from New York by either stopping in Washington DC or Chicago.
Overview and Contributions. Provenance games [20], a game-theoretical formalization of provenance for first-order (FO) queries, also supports queries with negation. However, the approach is computationally expensive, because it requires instantiation of a provenance graph explaining all answers and missing answers. For instance, the provenance graph produced by this approach for our toy example already contains more than 64 (=4^3) nodes (i.e., only counting nodes corresponding to rule derivations), because there are 4^3 ways of binding values from $adom(I)=\{c,n,s,w\}$ to the 3 variables $(X,Y,$ and $Z)$ of the rule $r_1$. Typically, most of the nodes will not end up being part of the explanation for the user’s provenance question. To efficiently compute the explanation, we introduce a new Datalog program which computes part of the provenance graph of an explanation bottom-up. Evaluating this program over instance $I$ returns the edge relation of an explanation.

The main driver of our approach is a rewriting of Datalog rules that captures successful and failed rule derivations. This rewriting replaces the rules of the program with so-called firing rules. Firing rules for positive queries were first introduced in [19]. These rules are similar to other query instrumentation techniques that have been used for provenance capture such as the rewrite rules of Perm [13]. One of our major contributions is to extend this concept for negation and failed rule derivations which is needed to support Datalog with negation and missing answers. Firing rules provide sufficient information for constructing explanations. However, to make this approach efficient, we need to avoid capturing rule derivations that will not contribute an explanation, i.e., they are not connected to the nodes corresponding to the provenance question in the provenance graph. We achieve this by propagating information from the user’s provenance question throughout the query to prune rule derivations early on 1) if they do not agree with the constants in the question or 2) if we can determine that based on their success/failure status they cannot be part of the explanation. For instance, in our running example, $Q(n,s)$ can only be connected to successful derivations of the rule $r_1$ with $X=n$ and $Y=s$. We have presented a proof-of-concept version of our approach as a poster [21]. Our main contributions are:

- We introduce a provenance graph model for full first-order (FO) queries, expressed as non-recursive Datalog queries with negation (or Datalog for short).
- We extend the concept of firing rules to support negation and missing answers.
- We present an efficient method for computing explanations to provenance questions. Unlike the solution in [20], our approach avoids unnecessary work by focusing the computation on relevant parts of the provenance graph.
- We prove the correctness of our algorithm that computes the explanation to a provenance question.
- We present a full implementation of our approach in the GProM [11] system. Using this system, we compile Datalog into relational algebra expressions, and translate the expressions into SQL code that can be executed by a standard relational database backend.

The remainder of this paper is organized as follows. We formally define the problem in Sec. II, discuss related work in Sec. III, and present our approach for computing explanations in Sec. IV. We then discuss our implementation (Sec. V), present experiments (Sec. VI), and conclude in Sec. VII.
We now formally define the problem addressed in this work: how to find the subgraph of a provenance graph for a given query (input program) \( P \) and instance \( I \) that explains existence/absence of a tuple-in/from the result of \( P \).

### A. Datalog

A Datalog program \( P \) consists of a finite set of rules
\[
   r_1 : R(X_1) := R_1(X_1), \ldots, R_n(X_n) \quad \text{where } X_j \text{ denotes a tuple of variables and/or constants.}
\]
We assume that the rules of a program are labeled \( r_1 \) to \( r_m \), \( R(X_j) = \text{the head of the rule, denoted } head(r_i) \), and \( R_1(X_1), \ldots, R_n(X_n) \) is the body (each \( R_j(X_j) \) is a goal). We use \( vars(r_i) \) to denote the set of variables in \( r_i \). In this paper, consider non-recursive Datalog with negation, so goals \( R_j(X_j) \) in the body are literals, i.e., atoms \( A(X_j) \) or their negation \( \neg A(X_j) \). Recursion is not allowed. All rules \( r \) of a program have to be safe, i.e., every variable in \( r \) must occur positively in \( r \)'s body (thus, head variables and variables in negated goals must also occur in a positive goal). For example, Fig. I shows a Datalog query with a single rule \( r_1 \). Here, \( head(r_1) = Q(X,Y) \) and \( vars(r_1) = \{ X,Y,Z \} \). The rule is safe since the head variables and the variables in the negated goal also occur positively in the body \((X,Y) \) in both cases). The set of relations in the schema over which \( P \) is defined is referred to as the extensional database (EDB), while relations defined through rules in \( P \) form the intensional database (IDB), i.e., the IDB relations \( P \) defines are labeled \( \text{and } I \text{ occurs as a set of } \). For an EDB or IDB predicate \( R \), we use \( P(I) \) to denote the instance of \( R \) computed by \( P \). We now formally define the problem addressed in this work: how to find the subgraph of a provenance graph for a given query (input program) \( P \) and instance \( I \) that explains existence/absence of a tuple-in/from the result of \( P \).

### B. Negation and Domains

To be able to explain why a tuple is missing, we have to enumerate all failed derivations of this tuple and, for such derivation, explain why it failed. As mentioned in Sec. I, the question is what is a feasible set of potential answers to be considered as missing. While the size of why-not provenance is typically infinite under the open world assumption, we have to decide how to bound the set of missing answers in the closed world assumption. We propose a simple, yet general, solution by assuming that each attribute of an IDB or EDB relation has an associated domain.

**Definition 1 (Domain Assignment).** Let \( S = \{ R_1, \ldots, R_n \} \) be a database schema where each \( R_j[\{A_1, \ldots, A_n \}] \) is a relation schema. Given an instance \( I \) of \( S \), a domain assignment \( dom \) is a function that associates with each attribute \( A \) a domain of values. We require \( dom(A) \supseteq \text{dom}(A) \).

In our approach, the user specifies each \( dom_r(A) \) as a query \( dom_r(A) \) that returns the set of admissible values for the domain of attribute \( A \). We provide reasonable defaults to avoid forcing the user to specify \( dom \) for every attribute, e.g., \( dom_r(A) = dom(A) \) for unspecified \( dom_r(A) \). These associated domains fulfill two purposes: 1) to reduce the size of explanations and 2) to avoid semantically meaningless answers. For instance, if there would exist another attribute \( Price \) in the relation \( Train \) in Fig. I, then \( dom_r(I) \) would also include all the values that appear in this attribute. Thus, some failed rule derivations for \( r_1 \) would assign prices to the variable representing intermediate stops. Different attributes may represent the same type of entity (e.g., \( from \), \( to \), \( City \) in our example) and, thus, it would make sense to use their combined domain values when constructing missing answers. For now, we leave it up to the user to specify attribute domains. Using techniques for discovering semantic relationships among attributes to automatically determine feasible attribute domains is an interesting avenue for future work.

When defining provenance graphs in the following, we are only interested in rule derivations that use constants from the associated domains of attributes accessed by the rule. Given a rule \( r \) and variable \( X \) used in this rule, let \( attr_r(r,X) \) denote the set of attributes that variable \( X \) is bound to in the body of the rule. For instance, in Fig. I, \( attr_r(r_1, Z) = \{ \text{Train, fromCity, Train, toCity} \} \). We say a rule derivation \( r(c_1, \ldots, c_n) \) is domain grounded iff \( c_i \in \bigcap_{A \in \text{attr}(r,c_i)} dom(A) \) for all \( i \in \{1, \ldots, n\} \).

### C. Provenance Graphs

Provenance graphs justify the existence or absence of a query result based on the success or failure of derivations using a query’s rules, respectively. They also explain how the existence or absence of tuples in the database caused derivations to succeed or fail, respectively. Here, we present a constructive definition of provenance graphs that provide this type of justification. Nodes in these graphs carry two types of labels: 1) a label that determines the node type (tuple, rule, or goal) and additional information, e.g., the arguments and rule identifier of a derivation; 2) the success/failure status of nodes.

**Definition 2 (Provenance Graph).** Let \( P \) be a first-order (FO) query. \( I \) database instance, \( dom_r(A) \) a domain assignment for \( I \), and \( L \) the domain containing all strings. The provenance graph \( PG(P,I) \) is a graph \((V,E,L,S)\) with nodes \( V \), edges \( E \), and node labeling functions \( L : V \rightarrow L \) and \( S : V \rightarrow \{ T,F \} \). We require that \( \forall v,v' \in V : L(v) = L(v') \rightarrow v = v' \). \( PG(P,I) \) is defined as follows:

- **Tuple nodes**: For each \( n \)-ary EDB or IDB predicate \( R \) and tuple \( c_1, \ldots, c_n \) of constants from the associated domains \((c_i \in dom_r(A_i))\), there exists a node \( v \) labeled \( R(c_1, \ldots, c_n) \). \( S(v) = T \iff R(c_1, \ldots, c_n) \in P(I) \) and \( S(v) = F \) otherwise.
- **Rule nodes**: For every successful domain grounded derivation \( r_1(c_1, \ldots, c_n) \), there exists a node \( v \) in \( V \) labeled \( r_1(c_1, \ldots, c_n) \) with \( S(v) = T \). For every failed domain grounded derivation \( r_1(c_1, \ldots, c_n) \) where \( \text{head}(r_1) \)
have considered derivations of variables and constants from the associated domain (provenance graphs. Such a set of tuples is called a 1-tuple and, thus, its node is connected to all failed derivations with this tuple node is connected to all successful derivations with the nodes.

Our provenance graphs model query evaluation by construction. A tuple node $R(t)$ is successful in $\mathcal{PG}(P,I)$ if $R(t) \in P(I)$. This is guaranteed, because each tuple defined as $\mathcal{S}(v)$ in a graph is uniquely identified by its label $L(v)$. Thus, common subexpressions are shared leading to more compact provenance graphs. For instance, observe that the node $g_1^2(n,s)$ is shared by two rule nodes in the explanation shown in Fig. 2.

D. Questions and Explanations

Recall that the problem we address in this work is how to explain the existence or absence of (sets of) tuples using provenance graphs. Such a set of tuples is called a provenance question (PQ) in this paper. The two questions presented in Example 1 use constants only, but we also support provenance queries with variables, e.g., for a question $Q(n,X)$ we would return all explanations for existing or missing tuples where the first attribute is $n$, i.e., why or why-not a city $X$ can be reached from New York with one transfer, but not directly. We say a tuple $t'$ of constants matches a tuple $t$ of variables and constants written as $t' \leq t$ if we can unify $t'$ with $t$, i.e., we can equate $t'$ with $t$ by applying a valuation that substitutes variables in $t$ with constants from $t'$.

Definition 3 (Provenance Question). Let $P$ be a query, $I$ an instance, and $Q$ an IDB predicate. A provenance question $PQ$ is an atom $Q(t)$ where $t = (c_1, \ldots, c_n)$ is a tuple consisting of variables and constants from the associated domain $(dom(Q.A)$ for each attribute $Q.A$). WHY $Q(t)$ and WHYNOT $Q(t)$ restrict the question to existing and missing tuples $t' \leq t$, respectively.

In Example 1, we have presented subgraphs of $\mathcal{PG}(P,I)$ as explanations for PQs, implicitly claiming that these subgraphs are sufficient for explaining the PQ. Below, we formally define this type of explanation.

Definition 4 (Explanation). The explanation $\text{EXPL}(P,Q(t),I)$ for $Q(t)$ (PQ) according to $P$ and $I$, is the subgraph of $\mathcal{PG}(P,I)$ containing only nodes that are connected to at least one node $Q(t')$ where $t' \leq t$. For $\text{WHY}$ $Q(t)$, only existing tuples $t'$ are considered to match $t$. For $\text{WHYNOT}$ $Q(t)$ only missing tuples are considered to match $t$.

Given this definition of explanation, note that 1) all nodes connected to a tuple node matching the PQ are relevant for computing this tuple and 2) only nodes connected to this node are relevant for the outcome. Consider $t'$ where $t' \not\leq t$ for a $Q(t')$ (PQ). If $Q(t') \in P(I)$, then all successful derivations with head $Q(t')$ justify the existence of $t'$ and these are precisely the rule nodes connected to $Q(t')$ in $\mathcal{PG}(P,I)$. If $Q(t') \not\in P(I)$, then all derivations with head $Q(t')$ have failed and are connected to $Q(t)$ in the provenance graph. Each such derivation is connected to all of its failed goals which are responsible for the failure. Now, if a rule body references IDB predicates, then the same argument can be applied to reason that all rules directly connected to these tuples explain why they (do not) exist. Thus, by induction, the explanation contains all relevant tuple and rule nodes that explain the PQ.

III. Related Work

Our provenance graphs have strong connections to other provenance models for relational queries, most importantly provenance games and the semiring framework, and to approaches for explaining missing answers.

Provenance Games. Provenance games [20] model the evaluation of a given query (input program) $P$ over an instance $I$ as a 2-player game in a way that resembles SLD(NF) resolution. If the position (a node in the game graph) corresponding to a tuple $t$ is won (the player starting in this position has a winning strategy), then $t \in P(I)$ and if the position is lost, then $t \not\in P(I)$. By virtue of supporting negation, provenance games can uniformly answer why and why-not questions for queries with negation. However, provenance games may be hard to comprehend for non-power users as they require some background in game theory to be understood, e.g., the won/lost status of derivations in a provenance game is contrary to what may be intuitively expected. That is, a rule node is lost if the derivation is successful. The status of rule nodes in our provenance graphs matches the intuitive expectation (e.g., the rule node is successful if the derivation exists). Köhler et al. [20] also present an algorithm that computes the provenance game for $P$ and $I$. However, this approach requires instantiation of the full game graph (which enumerates all existing and missing tuples) and evaluation of a recursive Datalog program over this graph using the well-founded semantics [11]. In constrast, our approach computes explanations that are succinct subgraphs containing only relevant provenance. We use bottom-up evaluation instrumented with firing rules to capture provenance. Furthermore, we enable the user to restrict provenance for missing answers.

Database Provenance. Several provenance models for database queries have been introduced in related work, e.g., see [7], [18]). The semiring annotation framework generalizes
these models for positive relational algebra (and, thus, positive non-recursive Datalog). In this model, tuples in a relation are annotated with elements from a commutative semiring \( \mathbb{N} \). An essential property of the \( K \)-relational model is that semiring \( \mathbb{N}[X] \), the semiring of \textit{provenance polynomials}, generalizes all other semirings. It has been shown in [20] that provenance games generalize \( \mathbb{N}[X] \) for positive queries and, thus, all other provenance models expressible as semirings. Since our graphs are equivalent to provenance games in the sense that there exist lossless transformations between both models (the discussion is beyond the scope of this paper), our graphs also encode \( \mathbb{N}[X] \). Provenance graphs which are similar to our graphs restricted to positive queries have been used as graph representations of semiring provenance (e.g., see [8], [9], [18]). Both our graphs and the boolean circuits representation of semiring provenance [9] explicitly share common subexpressions in the provenance. However, while these circuits support recursive queries, they do not support negation. Exploring the relationship of provenance graphs for queries with negation and m-semirings (semirings with support for set difference) is an interesting avenue for future work. Justifications for logic programs [23] are also closely related.

**Why-not and Missing Answers.** Approaches for explaining missing answers can be classified based on whether they explain a missing answer based on the query [3], [4], [6], [25] (i.e., which operators filter out tuples that would have contributed to the missing answer) or based on the input data [16], [17] (i.e., what tuples need to be inserted into the database to turn the missing answer into an answer). The missing answer problem was first stated for query-based explanations in the seminal paper by Chapman et al. [6]. Huang et al. [17] first introduced an instance-based approach. Since then, several techniques have been developed to exclude spurious explanations, to support larger classes of queries [16], and to support distributed Datalog systems in Y! [26]. The approaches for instance-based explanations (with the exception of Y!) have in common that they treat the missing answer problem as a view update problem: the missing answer is a tuple that should be inserted into a view corresponding to the query and this insert has to be translated into an insert into the database instance. An explanation is then one particular solution to this view update problem. In contrast to these previous works, our provenance graphs explain missing answers by enumerating all failed rule derivations that justify why the answer is not in the result. Thus, they are arguably a better fit for use cases such as debugging queries, where in addition to determining which missing inputs justify a missing answer, the user also needs to understand why derivations have failed. Furthermore, we do support queries with negation. Importantly, solutions for view update missing answer problems can be extracted from our provenance graphs. Thus, in a sense, provenance graphs with our approach generalize some of the previous approaches (for the class of queries supported, e.g., we do not support aggregation yet). Interestingly, recent work has shown that it may be possible to generate more concise summaries of provenance games [12], [24] which is particularly useful for negation and missing answers to deal with the potentially large size of the resulting provenance. Similarly, some missing answer approaches [16] use c-tables to compactly represent sets of missing answers. These approaches are complementary to our work.

**Computing Provenance Declaratively.** The concept of rewriting a Datalog program using firing rules to capture provenance as variable bindings of rule derivations was introduced by Köhler et al. [19] for provenance-based debugging of positive Datalog queries. These rules are also similar to relational implementations of provenance polynomials in Perm [13], LogicBlox [14], and Orchestra [15]. Zhou et al. [27] leverage such rules for the distributed ExSPAN system using either full propagation or reference based provenance. An extension of firing rules for negation is the main enabler of our approach.

### IV. Computing Explanations

Recall from Sec. I that our approach generates a new Datalog program \( GP(P, Q(t), I) \) by rewriting a given query (input program) \( P \) to return the edge relation of explanation \( EXP(P, Q(t), I) \) for a provenance question \( Q(t) \) (PQ). In this section, we explain how to generate the program \( GP(P, Q(t), I) \) using the following steps:

1. **We unify the input program** with the PQ by propagating the constants in \( t \) top-down throughout the program to be able to later prune irrelevant rule derivations.

2. **Afterwards, we determine for which nodes in the graph** we can infer their success/failure status based on the PQ. We model this information as annotations on heads and goals of rules and propagate these annotations top-down.

3. **Based on the annotated and unified version created in the previous steps, we generate firing rules** that capture variable bindings for successful and failed rule derivations.

4. **To be in the result of one of the firing rules obtained in the previous step is a necessary, but not sufficient, condition for the corresponding PG(P, I) fragment to be connected to a node matching the PQ.** To guarantee that only relevant fragments are returned, we introduce additional rules that check connectivity to confirm whether each fragment is connected.

5. **Finally, we create rules that generate the edge relation of the provenance graph** (i.e., \( EXP(P, Q(t), I) \)) based on the rule binding information that the firing rules have captured.

In the following, we will explain each step in detail and illustrate its application based on the question \( \text{WHY Q(n, s)} \) from Example 1, i.e., why is New York connected to Seattle via a train connection with one intermediate stop, but is not directly connected to Seattle.

#### A. Unify the Program with \( PQ \)

The node \( Q(n, s) \) in the provenance graph (Fig. 2) is only connected to rule derivations which return \( Q(n, s) \). For instance, if variable \( X \) is bound to another city \( x \) (e.g., Chicago) in a derivation of the rule \( r_1 \), then this rule cannot return the tuple \( (n, s) \). This reasoning can be applied recursively to replace variables in rules with constants. That is, we unify the rules in the program top-down with the PQ. This step may produce multiple duplicates of a rule with different bindings. We use superscripts to make explicit the variable binding used by a replica of a rule.

**Example 4.** Given the question \( \text{WHY Q(n, s)} \), we unify the single rule \( r_1 \) using the assignment \( (X=n, Y=s) \):

\[
r_1^{(X=n, Y=s)} : Q(n, s) :\neg T(n, Z), T(Z, s), \neg T(n, s)
\]
We may have to create multiple partially unified versions of a rule or an EDB atom. For example, to explore successful derivations of \( Q(n, s) \) we are interested in both train connections from New York to some city \((T(n, Z))\) and from any city to Seattle \((T(Z, s))\). Furthermore, we need to know whether there is a direct connection from New York to Seattle \((T(n, s))\). The general idea of this step is motivated by \cite{2} which introduced “magic sets”, an approach for rewriting logical rules to cut down irrelevant facts by using additional predicates called “magic predicates”. Similar techniques exist in standard relational optimization under the name of predicate move around. We use the technique of propagating variable bindings in the query to restrict the computation based on the user’s interest. This approach is correct because if we bind a variable in the head of a rule, then only rule derivations that agree with this binding can derive tuples that agree with this binding. Based on this unification step, we know which bindings may produce fragments of \( PG(P, I) \) that are relevant for explaining the PQ. The algorithm implementing this step is given as Algorithm 1.

\[ \text{Algorithm 1 Unify Program With PQ} \]

1: \text{procedure} \text{UNIFYPROGRAM}(P, Q(t))
2: \hspace{1em} \text{todo} \leftarrow \{Q(t)\}
3: \hspace{1em} \text{done} \leftarrow \{\}
4: \hspace{1em} P_{t} = \[]
5: \hspace{1em} \text{while todo \neq \[] do}
6: \hspace{2em} a \leftarrow \text{POP(todo)}
7: \hspace{2em} \text{INSERT(done, a)}
8: \hspace{2em} \text{rules} \leftarrow \text{GETRULESFORATOM}(P, a)
9: \hspace{2em} \text{for all} \ r \in \text{rules do}
10: \hspace{3em} \text{unRule} \leftarrow \text{UNIFYRULE}(r, a)
11: \hspace{2em} P_{t} \leftarrow P_{t} :: \text{unRule}
12: \hspace{2em} \text{for all} \ g \in \text{body(unRule)} \text{ do}
13: \hspace{3em} \text{if} \ g \notin \text{done} \text{ then} \todo \leftarrow \todo :: g
14: \hspace{1em} \text{return} \ P_{t}

\[ \text{Algorithm 2 Success/Failure Annotations} \]

1: \text{procedure} \text{ANNOTPROGRAM}(P_{A}, Q(t))
2: \hspace{1em} \text{state} \leftarrow \text{typeof}(Q(t))
3: \hspace{1em} \text{todo} \leftarrow \{Q(t)_{\text{state}}\}
4: \hspace{1em} \text{done} \leftarrow \{\}
5: \hspace{1em} P_{A} = \[]
6: \hspace{1em} \text{while todo \neq \[] do}
7: \hspace{2em} a \leftarrow \text{POP(todo)}
8: \hspace{2em} \text{state} \leftarrow \text{typeof}(a)
9: \hspace{2em} \text{INSERT(done, a)}
10: \hspace{2em} \text{rules} \leftarrow \text{GETUNRULESFORATOM}(P, a)
11: \hspace{2em} \text{for all} \ r \in \text{rules do}
12: \hspace{3em} \text{annotRule} \leftarrow \text{ANNOTRULE}(r, \text{state})
13: \hspace{2em} P_{A} \leftarrow P_{A} :: \text{annotRule}
14: \hspace{2em} \text{for all} \ g \in \text{body(annotRule)} \text{ do}
15: \hspace{3em} \text{if} \ \text{state} = \text{F} \text{ then}
16: \hspace{4em} \text{newstate} \leftarrow \text{F/T}
17: \hspace{3em} \text{else}
18: \hspace{4em} \text{newstate} \leftarrow \text{state}
19: \hspace{4em} \text{if ISNEGATED}(g) \text{ then}
20: \hspace{5em} \text{newstate} \leftarrow \text{SWITCHSTATE(state)}
21: \hspace{4em} \text{if} \ g^{\text{newstate}} \notin \text{done} \land \text{isIDB}(g) \text{ then}
22: \hspace{5em} \text{todo} \leftarrow \text{todo} :: g^{\text{newstate}}
23: \hspace{2em} \text{for all} \ r \in P_{A} \text{ do}
24: \hspace{3em} \text{if} \ \text{typeof}(r) = \text{F/T} \text{ then}
25: \hspace{4em} P_{A} \leftarrow \text{REMOVEANNOTATEDRULES}(P_{A}, r, \{\text{F, T}\})
26: \hspace{1em} \text{return} \ P_{A}

We now propagate the annotations of the goals in \( r_{1} \) throughout the program. That is, for any goal that is an IDB predicate, we propagate its annotation to the head of all rules deriving the goal’s predicate and, then, propagate these annotations to the corresponding rule bodies. Note that the inverted annotation is propagated for negated goals. For instance, if \( T \) would be an IDB predicate, then the annotation on the goal \( \neg T(n, s) \) would be propagated as follows. We would annotate the head of all rules deriving \( T(n, s) \) with \( F \), because \( Q(n, s) \) can only exist if \( T(n, s) \) does not exist.

\[ \text{Example 5. Continuing with our running example question WHY} Q(n, s), \text{ we know that} Q(n, s) \text{ is successful because the tuple is in the result (Fig. 1). This implies that only successful rule nodes and their successful goal nodes can be connected to this tuple node. Note that this annotation does not imply that the rule} r_{1} \text{ would be successful for every} Z \text{ (i.e., every intermediate stop between New York and Seattle). It only indicates that it is sufficient to focus on successful rule derivations since failed ones cannot be connected to} Q(n, s). \]

\[ r_{1}^{(X=n, Y=s), T} : Q(n, s)^{T} \vdash \neg T(n, Z)^{T}, T(Z, s)^{T}, \neg T(n, s)^{T} \]

Partially unified atoms (such as \( T(n, Z) \)) may occur in both negative and positive goals of the rules of the program. We denote such atoms using a \( /T \) annotation. The use of these annotations will become more clear in the next subsection when we introduce firing rules. The pseudocode for the algorithm that determines these annotations is given as Algorithm 2. In short:

1) Annotate the head of all rules deriving tuples matching the question with \( T \) (why) or \( F \) (why-not).
2) Repeat the following steps until a fixpoint is reached:
   a) Propagate the annotation of a rule head to goals in the rule body as follows: propagate \( T \) for \( T \) annotated heads and \( /T \) for \( F \) annotated heads.
   b) For each annotated goal in the rule body, propagate its annotation to all rules that have this atom in the head. For negated goals, unless the annotation is \( /T \), we propagate the inverted annotation (e.g., \( F \) for \( T \)) to the head of rules deriving the goal’s predicate.

C. Creating Firing Rules

To be able to compute the relevant subgraph of \( PG(P, I) \) (the explanation) for the provenance question \( PQ \), we need to
determine successful and/or failed rule derivations. Each rule
derivation paired with the information whether it is successful
over the given database instance (and which goals are failed
in case it is not successful) corresponds to a certain subgraph.
Successful rule derivations are always part of \( PG(P, I) \) for
a given query (input program) \( P \) whereas failed rule derivations
only appear if the tuple in the head failed, i.e., there are no
successful derivations of any rule with this head. To capture
the variable bindings of successful/failed rule derivations, we
create "firing rules". For successful rule derivations, a firing
rule consists of the body of the rule (but using the firing version
of each predicate in the body) and a new head predicate that
contains all variables used in the rule. In this way, the firing
rule captures all the variable bindings of a rule derivation.
Furthermore, for each IDB predicate \( \bar{R} \) that occurs as a head
of a rule \( r \), we create a firing rule that has the firing version
of predicate \( \bar{R} \) in the head and a firing version of the rules
\( r \) deriving the predicate in the body. For EDB predicates, we
create firing rules that have the firing version of the predicate
in the head and the EDB predicate in the body.

Example 6. Consider the annotated program in Example 5 for
the question \( \text{WHY}\, Q(n, s) \). We generate the firing rules shown
in Fig. 4. The firing rule for \( r_1^{(X=n, Y=s)} \) (the second rule
from the top) is derived from the rule \( r_1 \) by adding \( Z \) (the only
existential variable) to the head, renaming the head predicate
as \( F_{r_1,T} \) and replacing each goal with its firing version (e.g.,
\( F_{r_1,T} \) for the two positive goals and \( F_{r_1,F} \) for the negated goal).
Note that negated goals are replaced with firing rules that have
inverted annotations (e.g., the goal \( \neg T(n, s) \) is replaced with
\( F_{r_1,F}(n, s) \)). Furthermore, we introduce firing rules for EDB
tuples (the three rules from the bottom in Fig. 4).

As mentioned in Sec. III, firing rules for successful rule
derivations have been used for declarative debugging of positive
Datalog programs \[19\] and, for non-recursive queries, are
essentially equivalent to rewrite rules that instrument a query to
compute provenance polynomials \[1, 13\]. We extend firing
rules to support queries with negation and capture missing
answers. To construct a \( PG(P, I) \) fragment corresponding
to a missing tuple, we need to find failed rule derivations
with the tuple in the head and ensure that no successful
derivations exist with this head (otherwise, we may capture
irrelevant failed derivations of existing tuples). In addition,
we need to determine which goals are failed for each failed
rule derivation because only failed goals are connected to the
node representing the failed rule derivation in the provenance
graph. To capture this information, we add additional boolean
variables - \( V_i \) for goal \( g_i \) - to the head of a firing rule that
record for each goal whether it failed or not. The body of
a firing rule for failed rule derivations is created by replacing
every goal in the body with its \( F/T \) firing version, and adding
the firing version of the negated head to the body (to ensure that
only bindings for missing tuples are captured). Firing rules
capturing failed derivations use the \( F/T \) firing versions of
their goals because not all goals of a failed derivation have
to be failed and the failure status determines whether the
corresponding goal node is part of the explanation. A firing
rule capturing missing IDB or EDB tuples may not be safe,
which means that only bindings for missing tuples are captured). Firing rules
capturing failed derivations use the \( F/T \) firing versions of
their goals because not all goals of a failed derivation have
to be failed and the failure status determines whether the
corresponding goal node is part of the explanation. A firing
rule capturing missing IDB or EDB tuples may not be safe,
which means that only bindings for missing tuples are captured). Firing rules
capturing failed derivations use the \( F/T \) firing versions of
their goals because not all goals of a failed derivation have
to be failed and the failure status determines whether the
corresponding goal node is part of the explanation. A firing
rule capturing missing IDB or EDB tuples may not be safe,
which means that only bindings for missing tuples are captured). Firing rules
capturing failed derivations use the \( F/T \) firing versions of
their goals because not all goals of a failed derivation have
to be failed and the failure status determines whether the
corresponding goal node is part of the explanation. A firing
rule capturing missing IDB or EDB tuples may not be safe,
which means that only bindings for missing tuples are captured). Firing rules
capturing failed derivations use the \( F/T \) firing versions of
their goals because not all goals of a failed derivation have
to be failed and the failure status determines whether the
corresponding goal node is part of the explanation. A firing
rule capturing missing IDB or EDB tuples may not be safe,
which means that only bindings for missing tuples are captured). Firing rules
capturing failed derivations use the \( F/T \) firing versions of
their goals because not all goals of a failed derivation have
to be failed and the failure status determines whether the
corresponding goal node is part of the explanation. A firing
rule capturing missing IDB or EDB tuples may not be safe,
### Algorithm 3: Create Firing Rules

1. `procedure CREATEFIRINGRULES(P_A, Q(t))`
2. \[ \text{P}_{\text{Fire}} \leftarrow {} \]
3. \[ \text{state} \leftarrow \text{typoof}(Q(t)) \]
4. \[ \text{todo} \leftarrow [Q(t)\text{state}] \]
5. \[ \text{done} \leftarrow \{ \} \]
6. \[ \text{while } \text{todo} \neq \{ \} \text{ do} \]
7. \[ R(t)^a \leftarrow \text{POP(todo)} \]
8. \[ \text{INSERT} \text{(done)}; R(t)^a \]
9. \[ \text{if } \text{isEDB}(R) \text{ then} \]
10. \[ \text{CREATEFIRINGRULE} (P_{\text{Fire}}, R(t)^a) \]
11. \[ \text{else} \]
12. \[ \text{CREATEIDBNEGRULE} (P_{\text{Fire}}, R(t)^{\neg}) \]
13. \[ \text{rules} \leftarrow \text{GETRULES}(R(t)^a) \]
14. \[ \text{for all } r \in \text{rules} \text{ do} \]
15. \[ \text{args} \leftarrow \text{vars(head(r))} \]
16. \[ \text{CREATEIDBP} (P_{\text{Fire}}, R(t)^a, r, \text{args}) \]
17. \[ \text{CREATEFIRINGRULE} (P_{\text{Fire}}, R(t)^a, r, \text{args}) \]
18. \[ \text{return } P_{\text{Fire}} \]

---

### Algorithm 4: Create Firing Rules Subprocedures

1. `procedure CREATEEDBFRIRINGRULE(P_{\text{Fire}}, R(t)^a)`
2. \[ [X_1, \ldots, X_n] \leftarrow \text{vars}(t) \]
3. \[ r_T \leftarrow \text{F}_{\text{ext}}(t) : \text{R}(t) \]
4. \[ r_F \leftarrow \text{F}_{\text{ext}}(t) : \text{dom}(X_1), \ldots, \text{dom}(X_n), \neg \text{R}(t) \]
5. \[ r_{F/T^{-1}} \leftarrow \text{F}_{\text{ext}}(t, \text{true}) \]
6. \[ r_{F/T-2} \leftarrow \text{F}_{\text{ext}}(t, \text{false}) \]
7. \[ \text{if } \sigma = T \text{ then} \]
8. \[ P_{\text{Fire}} \leftarrow P_{\text{Fire}} :: r_T \]
9. \[ \text{else if } \sigma = F \text{ then} \]
10. \[ P_{\text{Fire}} \leftarrow P_{\text{Fire}} :: r_F :: r_{F/T-1} :: R/T^{-2} \]
11. \[ \text{else} \]
12. \[ P_{\text{Fire}} \leftarrow P_{\text{Fire}} :: r_{F/T} :: r_{F/T} :: r_{F/T-1} :: R/T^{-2} \]

---

**York** is missing. This causes the third goal of \( r_1 \) to succeed for any derivation where \( X=s \) and \( Y=n \). For each partially unified EDB atom annotated with \( F/T \), we create four rules: one for existing tuples (e.g., \( F_{T,T} (s,Z) : T(s,Z) \)), one for the failure case (e.g., \( F_{T,F} (s,Z) : \neg \text{dom}(T(s,Z)) \)), and two for the \( F/T \) firing version. For the failure case, we use predicate \( \text{dom}(T) \) to ensure that the head \( r \) of the associated domains of the attributes these variables are bound to and do not exist in \( R \). This is achieved by adding goals \( \text{dom}(X_i) \) as explained in Example 7.

**Rules.** Consider a rule \( r : R(t) : g_1(X_1), \ldots, g_n(X_n) \). If the head of \( r \) is annotated with \( T \), then we create a rule with head \( F_{T,T}(X) \) where \( X = \text{vars}(r) \) and the same body as \( r \) except that each goal is replaced with its firing version with appropriate annotation (e.g., \( T \) for positive goals). For rules annotated with \( F \) or \( F/T \), we create one additional rule with head \( F_{T,F}(X,V) \) where \( X \) is defined as above, and \( V \) contains \( V_i \) if the \( i^{th} \) goal of \( r \) is positive and \( V_i \) otherwise. The body of this rule contains the \( F/T \) version of every goal in \( r \)'s body plus an additional goal \( F_{ext} \) to ensure that the head atom is failed. As an example for this type of rule, consider the third rule from the top in Fig. 5.

**EDB atoms.** For an EDB atom \( R(t)^T \), we use procedure CREATEEDBFIRINGRULE to create one rule \( F_{ext}(t) : R(t)^T \) that returns tuples from relation \( R \) that matches \( t \). For missing tuples \( (R(t)^F) \), we extract all variables from \( t \) (some arguments may be constants propagated during unification) and create a rule that returns all tuples that can be formed from values of the associated domains of the attributes these variables are bound to and do not exist in \( R \). This is achieved by adding goals \( \text{dom}(X_i) \) as explained in Example 7.

**IDB atoms.** For each rule \( r \) with head \( R(t) \), we create a rule \( F_{ext}(t) : F_{ext}(X) \) where \( X \) is the concatenation of \( t \) with all existential variables from the body of \( r \). IDB atoms with \( F \) or \( F/T \) annotations are handled in the same way as EDB atoms with these annotations. For each \( R(t)^F \), we create a rule with \( \neg F_{ext}(t) \) in the body using the associated domain queries to restrict variable bindings. For \( R(t)^F/T \), we add two additional rules as shown in Fig. 5 for EDB atoms.
Theorem 1 (Correctness of Firing Rules). Let \( P \) be an input program, \( r \) denote a rule of \( P \) with \( \nu \) goals, and \( P_{\text{Fire}} \) be the firing version of \( P \). We use \( r(t) \models P(I) \) to denote that the rule derivation \( r(t) \) is successful in the evaluation of program \( P \) over \( I \). The firing rules for \( P \) correctly determine existence of tuples, successful rule derivations, and failed rule derivations for missing answers:

- \( F_{R,T}(t) \in P_{\text{Fire}}(I) \iff R(t) \in P(I) \)
- \( F_{R,F}(t) \in P_{\text{Fire}}(I) \iff R(t) \notin P(I) \)
- \( F_{T,F}(t) \in P_{\text{Fire}}(I) \iff r(t) \not\models P(I) \)
- \( F_{T,T}(t) \in P_{\text{Fire}}(I) \iff r(t) \models P(I) \) and for \( i \in \{1, \ldots, m\} \) we have that \( V_i \) is false iff \( i^{th} \) goal fails in \( r(t) \).

**Proof:** We prove Theorem 1 by induction over the structure of a program. For the base case, we consider programs of “depth” 1, i.e., only EDB predicates are used in rule bodies. Then, we prove correctness for programs of depth \( n + 1 \) based on the correctness of programs of depth \( n \). We define the depth \( d \) of predicates, rules, and programs as follows: 1) for all EDB predicates \( R \), we define \( d(R) = 0 \); 2) for an IDB predicate \( R \), we define \( d(R) = \max_{r(h) = R} d(r) \), i.e., the maximal depth among all rules \( r \) with head \( h = R \); 3) the depth of a rule \( r \) is \( d(r) = \max_{R \subseteq \text{body}(r)} d(R) + 1 \), i.e., the maximal depth of all predicates in its body plus one; 4) the depth of a program \( P \) is the maximum depth of its rules: \( d(P) = \max_{r \in P} d(r) \).

### Example 8. Consider the firing rules for \( Q(n, s) \) shown in Fig. 4. The corresponding rules with connectivity checks are shown in Fig. 6. All the rule nodes corresponding to \( F_{R,T}(n, s, Z) \) are guaranteed to be connected to the \( PQ \) node(s) in the explanation. Thus, to guarantee that only nodes connected to the \( PQ \) node(s) are returned, we have to check whether they are actually connected.

**D. Connectivity Joins**

To be in the result of firing rules is a necessary, but not sufficient, condition for the corresponding rule node to be connected to a \( PQ \) node in the explanation. Thus, to guarantee that only nodes connected to the \( PQ \) node(s) are returned, we have to check whether they are actually connected.

Algorithm 5 Add Connectivity Joints

1: procedure ADDCONNECTIVITYRULES \((P_{\text{Fire}}, Q(t))\)
2: \( P_{\text{FC}} \leftarrow \emptyset \)
3: paths \leftarrow PATHSTARTINGIN \((P_{\text{Fire}}, Q(t))\)
4: for all \( p \in \text{paths} \) do
5: \( p \leftarrow \text{FILTERRULENODES}(p) \)
6: for all \( e = (n, X_1), \ldots, (n, X_n) \) do \( p \) do
7: \( \text{goals} \leftarrow \text{GETMATCHINGGOALS}(e) \)
8: for all \( g_k \in \text{goals} \) do
9: \( g_{\text{new}} \leftarrow \text{UNIFYHEAD}(F_{R_1, \sigma_1}(t_2), g_k, F_{R_2, \sigma_2}(t_2)) \)
10: \( r_{\text{new}} \leftarrow \text{FC}_1(t_1^1, \sigma_1)(t_2) : \text{-body}(F_{R_1, \sigma_2}(t_2)), g_{\text{new}} \)
11: \( P_{\text{FC}} \leftarrow P_{\text{FC}} \cup r_{\text{new}} \)
12: return \( P_{\text{FC}} \)

Algorithm 5 traverses the query’s rules starting from the \( PQ \) to find all combinations of rules \( r_i, r_j \) such that the head of \( r_j \) can be unified with a goal in the body of \( r_i \). We use the subprocedure FILTERRULENODES to prune rules containing only constants. For each such pair \((r_i, r_j)\) where the head of \( r_j \) corresponds to the \( k^{th} \) goal of the body of \( r_i \), we create a rule \( FC_{r_2, r_1}(X) \) as follows. We unify the variables of
the $k^{th}$ goal in the firing rule for $r_i$ with the head variables of the firing rule for $r_j$. All remaining variables of $r_i$ are renamed to avoid name clashes. We then add the unified head of $r_i$ to the body of $r_j$. Effectively, these rules check one hop at a time whether rule nodes in the provenance graph are connected to the nodes matching the PQ.

E. Computing the Edge Relation

The program created so far captures all the information needed to generate the edge relation of the graph for the PQ. To compute the edge relation, we use Skolem functions to create node identifiers. The identifier of a node captures the type of the node (tuple, rule, or goal), assignments from variables to constants, and the success/failure status of the node, e.g., a tuple node $T(n,s)$ that is successful would be represented as $f_T^e(n,s)$. Each rule firing corresponds to a fragment of $\mathcal{PG}(P,I)$. For example, one such fragment is shown in Fig. 7 (left). Such a substructure is created through a set of rules:

- One rule creating edges between tuple nodes for the head predicate and rule nodes
- One rule for each goal connecting a rule node to that goal node (for failed rules only the failed goals are connected)
- One rule creating edges between each goal node and the corresponding EDB tuple node

The pseudocode for creating rules that return the edge relation is provided as Algorithm 6.

Example 9. Consider the firing rules with connectivity joins from Example 8. Some of the rules for creating the edge relation for the explanation sought by the user are shown in Fig. 7 (on the right side). For example, each edge connecting the tuple node $Q(n,s)$ to a successful rule node $r_1(n,s,Z)$ is created by the top-most rule, the second rule creates an edge between $r_1(n,s,Z)$ and $g_1^r(n,Z)$, and so on.

F. Correctness

We now state correctness of our approach for computing an explanation $\text{Exp}(P,Q(t),I)$ for a provenance question.

Theorem 2 (Correctness). The result of a Datalog program $\mathcal{GP}(P,Q(t),I)$ that rewriting a query (input program) $P$ over instance $I$ is the edge relation of $\text{Exp}(P,Q(t),I)$.

Proof: For Theorem 2, we prove that 1) only edges from $\mathcal{PG}(P,I)$ are returned by the program $\mathcal{GP}(P,Q(t),I)$ and 2) the program returns precisely the set of edges of explanation $\text{Exp}(P,Q(t),I)$.

1. The constant values used as variable binding by the rules creating edges in $\mathcal{GP}(P,Q(t),I)$ are either constants that occur in the PQ or the result of rules which are evaluated over the instance $I$. Since only the rules for creating the edge relation create new values (through Skolem functions), it follows that any constant used in constructing a node argument exists in the associated domain. Recall that the $\mathcal{PG}(P,I)$ only contains nodes with arguments from the associated domain. Any edge returned by $\mathcal{GP}(P,Q(t),I)$ is strictly based on the structure of the input program and connects nodes that agree on variable bindings. Thus, each edge produced by $\mathcal{GP}(P,Q(t),I)$ will be contained in $\mathcal{PG}(P,I)$.

Algorithm 6 Create Edge Relation

1: procedure CREATEEDGERELATION($P_{FC}, Q(t)$)
2: $P_M \leftarrow []$
3: todo $\leftarrow \{Q(t)\}$
4: done $\leftarrow \{\}$
5: while todo $\neq []$ do
6: $R(t)^o$ $\leftarrow$ POP(todo)
7: if $R(t)^o$ $\in$ done then
8: continue
9: done $\leftarrow$ INSERT(done, $R(t)^o$)
10: rules $\leftarrow$ GETRULES($R(t)^o$)
11: for all $r \in$ rules do
12: if $\text{isEDB}(r)$ then
13: if $\sigma = T$ then
14: $r_{new} \leftarrow \text{edge}(f_T^e(t'), f_T^e(t')) : F_T^e(args)$
15: else $r_{new} \leftarrow \text{edge}(f_T^e(t'), f_T^e(t')) : F_T^e(args)$
16: else $args_{new} \leftarrow b_1, b_2, ..., b_n$
17: if $\text{isNEGATED}(g)$ then
18: $r_{new} \leftarrow \text{edge}(f_T^e(t'), f_T^e(t')) : F_T^e(args, args_{new})$
19: else
20: $r_{new} \leftarrow \text{edge}(f_T^e(t'), f_T^e(t')) : F_T^e(args, args_{new})$
21: $P_M \leftarrow P_M \cup r_{new}$
22: $\sigma_{r} = \text{SWITCHSTATE}(\sigma)$
23: $r_{new} \leftarrow \text{edge}(f_T^e(t), f_T^e(t)) : F_T^e(args, t, ..., t)$
24: $P_M \leftarrow P_M \cup r_{new}$
25: for all $g(t') \in \text{body}(r)$ do
26: if $\text{isNEGATED}(g)$ then
27: $\sigma' = \text{SWITCHSTATE}(\sigma)$
28: else
29: $\sigma' = \sigma$
30: todo $\leftarrow$ todo $\cup$ $g(t')$
31: if $\sigma = T$ then
32: $r_{new} \leftarrow \text{edge}(f_T^e(args), f_T^e(t)) : F_T^e(args)$
33: else $args_{new} \leftarrow b_1, b_2, ..., b_n$
34: $r_{new} \leftarrow \text{edge}(f_T^e(t), f_T^e(t)) : F_T^e(args, args_{new})$
35: $P_M \leftarrow P_M \cup r_{new}$
36: return $P_M$

2. We now prove that the program $\mathcal{GP}(P,Q(t),I)$ returns precisely the set of edges of $\text{Exp}(P,Q(t),I)$. Assume that the $Q(t)$ (PQ) only uses constants (the extension to PQ which contains variables is immediate). Consider a rule of an input program of depth 1 (i.e., only EDB predicates in the body of rules). For such a rule node to be connected to the node $Q(t)$, its head variables have to be bound to $t$ (guaranteed by the unification step in Sec. IV-A). Since the firing rules are known to be correct, this guarantees that exactly the rule nodes connected to the PQ node are generated. The propagation
of this unification to the firing rules for EDB predicates is correct, because only EDB nodes agreeing with this binding can be connected to such a rule node. However, propagating constants is not sufficient since the firing rule for an EDB predicate (e.g., \( R \)) may return irrelevant tuples, i.e., tuples that are not part of any rule derivations for \( Q(t) \) (e.g., there may not exist EDB tuples for other goals in the rule which share variables with the particular goal using predicate \( R \)). This is checked by the connectivity joins (Sec. IV-D). If a tuple is returned by a connected firing rule, then the corresponding node is guaranteed to be connected to at least one rule node deriving \( PQ \). Note that this argument does not rely on the fact that predicates in the body of a rule are EDB predicates. Thus, we can apply this argument in a proof by induction to show that, given that rules of depth up to \( n \) only produce connected rule derivations, the same holds for rules of depth \( n + 1 \).

V. IMPLEMENTATION

We have implemented the approach presented in Sec. IV in our provenance middleware called GProM [1] that executes provenance requests using a relational database backend (shown in Fig. 8). The system was originally developed to support provenance requests for SQL. We have extended the system to support Datalog enriched with syntax for stating provenance questions. The user provides a why or why-not question and the corresponding Datalog query as input. Our system parses and semantically analyzes this input. Schema information is gathered by querying the catalog of the backend system parses and semantically analyzes this input. Schema information is gathered by querying the catalog of the backend system is not a co-author or indirect co-author (2 hops) of \( Y \).

We work with the TPC-H benchmark, we used the following database sizes: 10MB, 100MB, 1GB, and 10GB. All experiments were run on a machine with 2 x 3.3GHz AMD Opteron 4238 CPUs (12 cores in total) and 128GB RAM running Oracle Linux 6.4. We use the commercial DBMS X (name omitted due to licensing restrictions) as a backend. Unless stated otherwise, each experiment was repeated 100 times and we report the median runtime. We allocated a timeslot of 10 minutes for each run. Computations that did not finish in the allocated time are omitted from the graphs.

Workloads. We compute explanations for the queries in Fig. 10 over the datasets we have introduced. For the Datalog PROV-DB dataset, we consider: only2hop (\( r_1 \)) which is our running example query; \( X \) with \( Y \) not \( Z \) (\( r_2 \)) that returns authors that are direct co-authors of a certain person \( Y \), but not of “Svein Johannessen”; only3hop (\( r_3 \)) that returns pairs of authors \((X,Y)\) that are connected via a path of length 3 in the co-author graph where \( X \) is not a co-author or indirect co-author (2 hops) of \( Y \). For TPC-H, we consider: ordPriority (\( r_4 \)) which returns for each customer the priorities of her/his orders; ordDisc (\( r_5 \)) which returns customers and the discount rates of items in their orders; partNotAsia (\( r_6 \)) which finds parts that can be supplied from a country that is not in Asia.

Our translation of first-order (FO) queries (a program with a distinguished answer relation) to \( \mathcal{RA} \) is mostly standard. We first translate each rule into an algebra expression independently. Afterwards, we create expressions for IDB predicates as a union of the expressions for all rules with the predicates in the head. Finally, the algebra expressions for individual IDB predicates are connected into a single graph by replacing references to IDB predicates with their algebraic translation.

Example 10. Consider the translation of the rule \( r_1 \) from Fig. 1. The \( \mathcal{RA} \) graph for \( r_1 \) is shown in Fig. 9. The translations of the first two goals are joined to compute the variable bindings for the positive part of the query. The negated goal is translated into a set difference between the positive part projected on \( X,Y \) and relation \( Train \). The remaining three operators (from the left) join the positive with the negative part, project on the head variables, and remove duplicates.

VI. EXPERIMENTS

We evaluate the performance of our solution over a co-author graph relation extracted from DBLP (http://www.dblp.org/) as well as over the TPC-H decision support benchmark (http://www.tpc.org/tpch/default.asp). We compare our approach for computing explanations with the approach introduced for provenance games [20]. We call the provenance game approach Direct Method (DM), because it directly constructs the full provenance graph. We have created subsets of the DBLP dataset with 100, 1K, 10K, 100K, 1M, and 8M co-author pairs (tuples). For the TPC-H benchmark, we used the following database sizes: 10MB, 100MB, 1GB, and 10GB. All experiments were run on a machine with 2 x 3.3GHz AMD Opteron 4238 CPUs (12 cores in total) and 128GB RAM running Oracle Linux 6.4. We use the commercial DBMS X (name omitted due to licensing restrictions) as a backend. Unless stated otherwise, each experiment was repeated 100 times and we report the median runtime. We allocated a timeslot of 10 minutes for each run. Computations that did not finish in the allocated time are omitted from the graphs.

1More examples for our method and installation guideline for GProM are available at https://github.com/IITDBGroup/gprom/wiki/datalog_prov.
Implementing DM. As introduced in Sec. I, DM has to instantiate a graph with $O(|\text{adom}(I)|^n)$ nodes where $n$ is the maximal number of variables in a rule. We do not have a full implementation of DM, but can compute a conservative lower bound for the runtime of the step constructing the game graph by executing a query that computes an $n$-way cross-product over the active domain. Note that the actual runtime will be much higher because 1) several edges are created for each rule binding (we underestimate the number of nodes of the constructed graph) and 2) recursive Datalog queries have to be evaluated over this graph using the well-founded semantics. The results for different instance sizes and number of variables are shown in Fig. 11. Even for only 2 variables, DM did not finish for datasets of more than 10K tuples within the allocated 10 min timeslot. For queries with more than 4 variables, DM did not even finish for the smallest dataset.

Why Questions. The runtime incurred for generating explanations for why questions over the queries $r_1$, $r_2$, $r_4$, and $r_5$ (Fig. 10) is shown in Fig. 12. For the evaluation, we consider the effect of the different binding patterns on performance. Fig. 12.c and 12.f show the variables bound by the PQs we have considered. Fig. 12.a and 12.b show the performance results for $r_1$ and $r_2$, respectively. We also show the number of rule nodes in the provenance graph for each binding pattern below the X axis. If only variable $X$ is bound ($\text{BindingXY}$), then the queries determine authors that occur together with the author with whom we have bound to $X$ in the query result. For instance, the explanation derived for only2hop with BindingX (Fig. 12.a) explains why persons are indirect, but not direct, co-authors of “Tore Risch”. If both $X$ and $Y$ are bound ($\text{BindingXY}$), then the explanation for $r_1$ and $r_2$ is limited to a particular indirect and direct co-author, respectively. The runtime for generating explanations using our approach exhibits roughly linear growth in the dataset size and dominates DM even for the small instances. Furthermore, Fig. 12.d and 12.e (for $r_4$ and $r_5$, respectively) show that our approach can handle queries with many variables where DM times out even for the smallest dataset we have considered. Binding one variable, e.g., BindingY, in queries $r_4$ and $r_5$ expresses a condition, e.g., $Y = '1-URGENT'$ in $r_1$ requires the order priority to be urgent. If both variables are bound, then the provenance question verifies the existence of orders for a certain customer (e.g., why “Customer16” has at least one urgent order). Runtimes exhibit the same trend as for the Datalog queries.

Why-not Provenance. We have queries $r_1$ and $r_2$ from Fig. 10 to evaluate the performance of computing explanations for failed derivations. When binding all variables in the PQ (BindingXY) with the information in Fig. 13.c, these queries check if a particular set of authors cannot appear together in the result. For instance, for only2hop ($r_1$) the query checks why “Tore Risch” is either not an indirect co-author or a direct
Fig. 14: Runtime - Why questions over queries with negation

We present a unified framework for explaining answers and non-answers over first-order (FO) queries. Our approach is based on the concept of firing rules that we extend to support negation and missing answers. Our efficient middleware implementation generates a Datalog program that computes the explanation for a provenance question and compiles this program into SQL. Our experimental evaluation demonstrates that by avoiding to generate irrelevant parts of the graph for the provenance question we can answer provenance questions over large instances. An interesting avenue for future work is to investigate summarized representation of provenance (e.g., in the spirit of [5], [10], [12], [24]) to deal with the large size of explanations for missing answers. Other topics of interest include considering integrity constraints in the provenance graph construction (e.g., rule derivations can never succeed if they violate integrity constrains), marrying the approach with ideas from missing answer approaches that only return one explanation that is optimal according to some criterion, and extending the approach for more expressive query languages (e.g., aggregation or non-stratified recursive programs).

VII. CONCLUSIONS

We present a unified framework for explaining answers and non-answers over first-order (FO) queries. Our approach is based on the concept of firing rules that we extend to support negation and missing answers. Our efficient middleware implementation generates a Datalog program that computes the explanation for a provenance question and compiles this program into SQL. Our experimental evaluation demonstrates that by avoiding to generate irrelevant parts of the graph for the provenance question we can answer provenance questions over large instances. An interesting avenue for future work is to investigate summarized representation of provenance (e.g., in the spirit of [5], [10], [12], [24]) to deal with the large size of explanations for missing answers. Other topics of interest include considering integrity constraints in the provenance graph construction (e.g., rule derivations can never succeed if they violate integrity constrains), marrying the approach with ideas from missing answer approaches that only return one explanation that is optimal according to some criterion, and extending the approach for more expressive query languages (e.g., aggregation or non-stratified recursive programs).

REFERENCES


