A SQL-Middleware Unifying *Why* and *Why-Not* Provenance for First-Order Queries

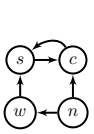
Seokki Lee* Sven Köhler† Bertram Ludäscher‡ Boris Glavic*
*Illinois Institute of Technology. {slee195@hawk.iit.edu, bglavic@iit.edu}
‡University of Illinois at Urbana-Champaign. {ludaesch@illinois.edu}
†University of California at Davis. {svkoehler@ucdavis.edu}

***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 [23] 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 $\neg Q$. 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.

$r_1 : \mathtt{Q}(X,Y) :- \mathtt{Train}(X,Z), \mathtt{Train}(Z,Y), \neg\mathtt{Train}(X,Y)$



Relation **Train**

fromCity	toCity
new york	washington dc
new york	chicago
chicago	seattle
seattle	chicago
washington dc	seattle

Result of query **Q**

X	Y
washington dc	chicago
new york	seattle
seattle	seattle
chicago	chicago

Fig. 1: Example train connection database and query

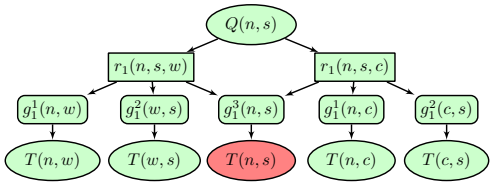


Fig. 2: Provenance graph explaining WHY $\mathtt{Q}(n,s)$

Example 1. *Consider the relation* `Train` *in Fig. 1 that stores train connections in the US. The Datalog rule* r_1 *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 $\mathtt{Q}(n,s)$*) with one intermediate stop but not directly or why it is not possible to reach New York from Seattle in the same fashion (*WHYNOT $\mathtt{Q}(s,n)$*).*

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 $\mathtt{Q}(n,s)$ and WHYNOT $\mathtt{Q}(s,n)$ 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 $\mathtt{Q}(n,s)$*. Seattle can be reached from New York by either stopping in Washington DC or Chicago*

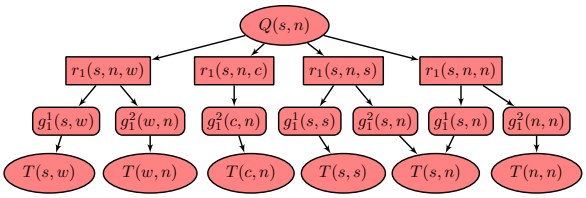


Fig. 3: Provenance graph explaning WHYNOT $\mathtt{Q}(s,n)$

and there is no direct connection between these two cities. These two options correspond to two successful derivations for rule r_1 with X=n, Y=s, and Z=w (or Z=c, respectively). In the provenance graph, there are two rule nodes *denoting these successful derivations of $\mathtt{Q}(n,s)$ by rule r_1. A derivation is successful if all goals in the body evaluate to true, i.e., a successful* rule node *is connected to successful* goal nodes *(e.g., r_1 is connected to g_1^1, the 1^{st} goal in the rule's body). A positive (negated) goal is successful if the corresponding tuple is (is not) in the database. Thus, a successful goal node is connected to the node corresponding to the existing (green) or missing (red) tuple justifying the goal, respectively.*

Supporting negation and missing answers is quite challenging, because we need to enumerate all potential ways of deriving a missing answer (or intermediate result corresponding to a negated subgoal) and explain why each of these derivations has failed. An important question in this respect is how to bound the set of missing answers to be considered. Under the open world assumption, provenance would be infinite. Using the closed world assumption, only values that exist in the database or are postulated by the query are used to construct missing tuples. As is customary in Datalog, we refer to this set of values as the active domain $adom(I)$ of a database instance I. We will revisit the assumption that all derivations with constants from $adom(I)$ are meaningful later on.

Example 3. *The explanation for* WHYNOT $\mathtt{Q}(s,n)$ *is shown in Fig. 3, i.e., why it is not true that New York is reachable from Seattle with exactly one transfer, but not directly. The tuple $\mathtt{Q}(s,n)$ is missing from the query result because all potential ways of deriving this tuple through the rule r_1 have failed. In this example, $adom(I)$=$\{c,n,s,w\}$ and, thus, there exist four failed derivations of $\mathtt{Q}(s,n)$ choosing either of these cities as the intermediate stop between Seattle and New York. A rule derivation fails if at least one goal in the body evaluates to false. In the provenance graph, only failed goals are connected to the failed rule derivations explaining missing answers. Failed positive goals in the body of a failed rule are explained by missing tuples (red* tuple nodes*). For instance, we cannot reach New York from Seattle with an intermediate stop in Washington DC (the first failed rule derivation from the left in Fig. 3) because there exists no connection from Seattle to Washington DC (a* tuple node $\mathtt{T}(s,w)$ *in red), and Washington DC to New York (a* tuple node $\mathtt{T}(w,n)$ *in red). Note that the successful goal $\neg\mathtt{T}(s,n)$ (there is no direct connection from Seattle to New York) does not contribute to the failure of this derivation and, thus, is not part of the explanation. A failed negated goal is explained by an existing tuple in the database. That is, if a tuple (s,n) would exist in the* `Train` *relation, then an additional failed* goal node $g_1^3(s,n)$ *would be part of the explanation and be connected to each failed rule derivation.*

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 generate a 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 to 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, $\mathtt{Q}(n,s)$ 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 [22]. 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 [1] system. Using this system, we compile Datalog into relational algebra expressions, and translate these 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.

II. Problem Definition

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 $r_i : \mathtt{R}(\vec{X}) :- \mathtt{R_1}(\vec{X_1}), \ldots, \mathtt{R_n}(\vec{X_n})$ where $\vec{X_j}$ denotes a tuple of variables and/or constants. We assume that the rules of a program are labeled r_1 to r_m. $\mathtt{R}(\vec{X})$ is the *head* of the rule, denoted $head(r_i)$, and $\mathtt{R_1}(\vec{X_1}), \ldots, \mathtt{R_n}(\vec{X_n})$ is the *body* (each $\mathtt{R_j}(\vec{X_j})$ 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 $\mathtt{R_j}(\vec{X_j})$ in the body are *literals*, i.e., atoms $\mathtt{A}(\vec{X_j})$ or their negation $\neg\mathtt{A}(\vec{X_j})$. 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. 1 shows a Datalog query with a single rule r_1. Here, $head(r_1)$ is $\mathtt{Q}(X, Y)$ and $vars(r_1)$ is $\{X, Y, Z\}$. The rule is safe since the head variables and the variables in the negated goal also occur positively in the body (X and 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 are those defined in the head of rules. We require that P has a distinguished IDB relation Q, called the *answer* relation. Given P and instance I, we use $P(I)$ to denote the result of P evaluated over I. Note that $P(I)$ includes the instance I, i.e., all EDB atoms that are true in I. For an EDB or IDB predicate R, we use $R(I)$ to denote the instance of R computed by P and $R(t) \in P(I)$ to denote that $t \in R(I)$ according to P.

We use $adom(I)$ to denote the active domain of instance I, i.e., the set of all constants that occur in I. Similarly, we use $adom(\mathtt{R.A})$ to denote the active domain of attribute A of relation $\mathtt{R}$. In the following, we make use of the concept of a rule derivation. A *derivation* of a rule r is an assignment of variables in r to constants from $adom(I)$. For a rule with n variables, we use $r(c_1, \ldots, c_n)$ to denote the derivation that is the result of binding X_i=c_i. We call a derivation *successful* wrt. an instance I if each atom in the body of the rule is true in I and *failed* otherwise.

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 each 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 $S = \{\mathtt{R_1}, \ldots, \mathtt{R_n}\}$ be a database schema where each $\mathtt{R_i}(\mathtt{A_1}, \ldots, \mathtt{A_m})$ is a relation schema. Given an instance I of S, a* domain assignment *dom is a function that associates with each attribute $\mathtt{R.A}$ a domain of values. We require $dom(\mathtt{R.A}) \supseteq adom(\mathtt{R.A})$.*

In our approach, the user specifies each $dom(\mathtt{R.A})$ as a query $dom_{\mathtt{R.A}}$ that returns the set of admissible values for the domain of attribute $\mathtt{R.A}$. We provide reasonable defaults to avoid forcing the user to specify dom for every attribute, e.g., $dom(\mathtt{R.A})$=$adom(\mathtt{R.A})$ for unspecified $dom_{\mathtt{R.A}}$. 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. 1, then $adom(I)$ 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., `fromCity` and `toCity` 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 $attrs(r, X)$ denote the set of attributes that variable X is bound to in the body of the rule. For instance, in Fig. 1, $attrs(r_1, Z)$=$\{\mathtt{Train.fromCity}, \mathtt{Train.toCity}\}$. We say a rule derivation $r(c_1, \ldots, c_n)$ is *domain grounded* iff $c_i \in \bigcap_{A \in attrs(r, X_i)} dom(A)$ 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) to derive it using a query's rules. 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 a database instance, dom a domain assignment for I, and $\mathbb{L}$ the domain containing all strings. The* provenance graph *$\mathcal{PG}(P, I)$ is a graph $(V, E, \mathcal{L}, \mathcal{S})$ with nodes V, edges E, and node labelling functions $\mathcal{L} : V \to \mathbb{L}$ and $\mathcal{S} : V \to \{T, F\}$ (true for T and false for F). We require that $\forall v, v' \in V : \mathcal{L}(v) = \mathcal{L}(v') \to v = v'$. $\mathcal{PG}(P, I)$ 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 ($c_i \in dom(\mathtt{R.A_i})$), there exists a node v labeled $R(c_1, \ldots, c_n)$. $\mathcal{S}(v) = T$ iff $R(c_1, \ldots, c_n) \in P(I)$ and $\mathcal{S}(v) = F$ otherwise.*
- ***Rule nodes:*** *For every successful domain grounded derivation $r_i(c_1, \ldots, c_n)$, there exists a node v in V labeled $r_i(c_1, \ldots, c_n)$ with $\mathcal{S}(v) = T$. For every failed domain grounded derivation $r_i(c_1, \ldots, c_n)$ where $head(r_i$*

$(c_1, \ldots, c_n)) \notin P(I)$, *there exists a node* v *as above but with* $\mathcal{S}(v) = F$. *In both cases,* v *is connected to the tuple node* $head(r_i(c_1, \ldots, c_n))$.

- ***Goal nodes:*** *Let* v *be the node corresponding to a derivation* $r_i(c_1, \ldots, c_n)$ *with* m *goals. If* $\mathcal{S}(v) = T$, *then for all* $j \in \{1, \ldots, m\}$, v *is connected to a goal node* v_j *labeled* g_i^j *with* $\mathcal{S}(v_j) = T$. *If* $\mathcal{S}(v) = F$, *then for all* $j \in \{1, \ldots, m\}$, v *is connected to a goal node* v_j *with* $\mathcal{S}(v_j) = F$ *if the* j^{th} *goal is failed in* $r_i(c_1, \ldots, c_n)$. *Each goal is connected to the corresponding tuple node.*

Our provenance graphs model query evaluation by construction. A tuple node $R(t)$ is successful in $\mathcal{PG}(P, I)$ iff $R(t) \in P(I)$. This is guaranteed, because each tuple built from values of the associated domain exists as a node v in the graph and its label $\mathcal{S}(v)$ is decided based on $R(t) \in P(I)$. Furthermore, there exists a successful rule node $r(\vec{c}) \in \mathcal{PG}(P, I)$ iff the derivation $r(\vec{c})$ succeeds for I. Likewise, a failed rule node $r(\vec{c})$ exists iff the derivation $r(\vec{c})$ is failed over I and $head(r(\vec{c})) \notin P(I)$. Fig. 2 and 3 show subgraphs of $\mathcal{PG}(P, I)$ for the query from Fig. 1. Since $\texttt{Q}(n, s) \in P(I)$ (Fig. 2), this tuple node is connected to all successful derivations with $\texttt{Q}(n, s)$ in the head which in turn are connected to goal nodes for each of the three goals of rule r_1. $\texttt{Q}(s, n) \notin P(I)$ (Fig. 3) and, thus, its node is connected to all failed derivations with $\texttt{Q}(s, n)$ as a head. Here, we have assumed that all cities can be considered as starting and end points of missing train connections, i.e., both $dom(\texttt{T.fromCity})$ and $dom(\texttt{T.toCity})$ are defined as $adom(\texttt{T.fromCity}) \cup adom(\texttt{T.toCity})$. Thus, we have considered derivations $r_1(s, n, Z)$ for $Z \in \{c, n, s, w\}$.

An important characteristic of our provenance graphs is that each node v in a graph is uniquely identified by its label $\mathcal{L}(v)$. Thus, common subexpressions are shared leading to more compact provenance graphs. For instance, observe that the node $g_1^3(n, s)$ 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 questions with variables, e.g., for a question $\texttt{Q}(n, X)$ 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 $t' \preccurlyeq t$ 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* $t = (v_1, \ldots, v_n)$ *is a tuple consisting of variables and constants from the associated domain (*$dom(\texttt{Q.A})$ *for each attribute* $\texttt{Q.A}$*).* WHY $Q(t)$ *and* WHYNOT $Q(t)$ *restrict the question to existing and missing tuples* $t' \preccurlyeq t$, *respectively.*

In Example 2 and 3, we have presented subgraphs of $\mathcal{PG}(P, I)$ as *explanations* for PQs, implicitly claiming that these subgraphs are sufficient for explaining the PQs. 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* $t' \preccurlyeq t$. *For* WHY $Q(t)$, *only existing tuples* t' *are considered to match* t. *For* 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 $t' \preccurlyeq t$ for a question $Q(t)$. 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 $Q(t') \notin P(I)$, 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 $t \notin P(I)$. 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$^\neg$ 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 and queries with negation.

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 K. An essential property of the K-relational model is that semiring $\mathbb{N}[X]$, the semiring of *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 [24] 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], [26] (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! [27]. 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 to 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], [25] 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. [28] 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 $\mathbb{GP}(P, Q(t), I)$ by rewriting a given query (input program) P to return the edge relation of explanation $\text{Expl}(P, Q(t), I)$ for a provenance question $Q(t)$. In this section, we explain how to generate the program $\mathbb{GP}(P, Q(t), I)$ using the following steps:

1. We unify the input program P 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 $\mathcal{PG}(P, I)$ 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., $\text{Expl}(P, Q(t), I)$) 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 $\text{Why}\,\mathtt{Q}(n, s)$ 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 $\mathtt{Q}(n, s)$ in the provenance graph (Fig. 2) is only connected to rule derivations which return $\mathtt{Q}(n, s)$. 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* $\text{Why}\,\mathtt{Q}(n, s)$*, we unify the single rule* r_1 *using the assignment* $(X{=}n, Y{=}s)$*:*

$$r_1^{(X=n,Y=s)} : \mathtt{Q}(n, s) :- \mathtt{T}(n, Z), \mathtt{T}(Z, s), \neg\, \mathtt{T}(n, s)$$

We may have to create multiple partially unified versions of a rule or an EDB atom. For example, to explore successful derivations of $\mathtt{Q}(n,s)$, we are interested in both train connections from New York to some city ($\mathtt{T}(n,Z)$) and from any city to Seattle ($\mathtt{T}(Z,s)$). Furthermore, we need to know whether there is a direct connection from New York to Seattle ($\mathtt{T}(n,s)$). The general idea of this step is motivated by [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 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 $\mathcal{PG}(P,I)$ that are relevant for explaining the PQ. The algorithm implementing this step is given in our accompanying technical report [21].

B. Add Annotations based on Success/Failure

For WHY and WHYNOT questions, we only consider tuples that are existing and missing, respectively. Based on this information, we can infer restrictions on the success/failure status of nodes in the provenance graph that are connected to PQ node(s) (belong to the explanation). We store these restrictions as annotations T, F, and F/T on heads and goals of rules. Here, T indicates that we are only interested in successful nodes, F that we are only interested in failed nodes, and F/T that we are interested in both. These annotations are determined using a top-down propagation seeded with the PQ.

Example 5. *Continuing with our running example question* WHY $\mathtt{Q}(n,s)$*, we know that* $\mathtt{Q}(n,s)$ *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 *would be successful for every* Z *(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* $\mathtt{Q}(n,s)$*.*

$$r_1^{(X=n,Y=s),T} : \mathtt{Q}(n,s)^T :- \mathtt{T}(n,Z)^T, \mathtt{T}(Z,s)^T, \neg\, \mathtt{T}(n,s)^T$$

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* $\mathtt{T}$ *would be an IDB predicate, then the annotation on the goal* $\neg\, \mathtt{T}(n,s)^T$ *would be propagated as follows. We would annotate the head of all rules deriving* $\mathtt{T}(n,s)$ *with* F*, because* $\mathtt{Q}(n,s)$ *can only exist if* $\mathtt{T}(n,s)$ *does not exist.*

Partially unified atoms (such as $\mathtt{T}(n,Z)$) may occur in both negative and positive goals of the rules of the program. We denote such atoms using a F/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 in our accompanying technical report [21]. 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 F/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 F/T, we propagate the inverted annotation (e.g., F for T) to the head of rules deriving the goal's predicate.

$$\mathtt{F}_{\mathtt{Q},\mathtt{T}}(n,s) :- \mathtt{F}_{r_1,\mathtt{T}}(n,s,Z)$$
$$\mathtt{F}_{r_1,\mathtt{T}}(n,s,Z) :- \mathtt{F}_{\mathtt{T},\mathtt{T}}(n,Z), \mathtt{F}_{\mathtt{T},\mathtt{T}}(Z,s), \mathtt{F}_{\mathtt{T},\mathtt{F}}(n,s)$$
$$\mathtt{F}_{\mathtt{T},\mathtt{T}}(n,Z) :- \mathtt{T}(n,Z)$$
$$\mathtt{F}_{\mathtt{T},\mathtt{T}}(Z,s) :- \mathtt{T}(Z,s)$$
$$\mathtt{F}_{\mathtt{T},\mathtt{F}}(n,s) :- \neg\, \mathtt{T}(n,s)$$

Fig. 4: Example firing rules for WHY $\mathtt{Q}(n,s)$

C. Creating Firing Rules

To be able to compute the relevant subgraph of $\mathcal{PG}(P,I)$ (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 $\mathcal{PG}(P,I)$ 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 R that occurs as a head of a rule r, we create a firing rule that has the firing version of predicate R in the head and a firing version of the rule 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* WHY $\mathtt{Q}(n,s)$*. We generate the firing rules shown in Fig. 4. The firing rule for* $r_1^{(X=n,Y=s),T}$ *(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* $\mathtt{F}_{r_1,\mathtt{T}}$*, and replacing each goal with its firing version (e.g.,* $\mathtt{F}_{\mathtt{T},\mathtt{T}}$ *for the two positive goals and* $\mathtt{F}_{\mathtt{T},\mathtt{F}}$ *for the negated goal). Note that negated goals are replaced with firing rules that have inverted annotations (e.g., the goal* $\neg\, \mathtt{T}(n,s)^T$ *is replaced with* $\mathtt{F}_{\mathtt{T},\mathtt{F}}(n,s)$*). Furthermore, we introduce firing rules for EDB tuples (the three rules from the bottom in Fig. 4)*

$$\mathtt{F}_{\mathtt{Q},\mathtt{F}}(s,n) :- \neg\, \mathtt{F}_{\mathtt{Q},\mathtt{T}}(s,n)$$
$$\mathtt{F}_{\mathtt{Q},\mathtt{T}}(s,n) :- \mathtt{F}_{\mathtt{r}_1,\mathtt{T}}(s,n,Z)$$
$$\mathtt{F}_{\mathtt{r}_1,\mathtt{F}}(s,n,Z,V_1,V_2,\neg V_3) :- \mathtt{F}_{\mathtt{Q},\mathtt{F}}(s,n), \mathtt{F}_{\mathtt{T},\mathtt{F}/\mathtt{T}}(s,Z,V_1), \mathtt{F}_{\mathtt{T},\mathtt{F}/\mathtt{T}}(Z,n,V_2), \mathtt{F}_{\mathtt{T},\mathtt{F}/\mathtt{T}}(s,n,V_3)$$
$$\mathtt{F}_{\mathtt{r}_1,\mathtt{T}}(s,n,Z) :- \mathtt{F}_{\mathtt{T},\mathtt{T}}(s,Z), \mathtt{F}_{\mathtt{T},\mathtt{T}}(Z,n), \mathtt{F}_{\mathtt{T},\mathtt{F}}(s,n)$$
$$\mathtt{F}_{\mathtt{T},\mathtt{F}/\mathtt{T}}(s,Z,true) :- \mathtt{F}_{\mathtt{T},\mathtt{T}}(s,Z)$$
$$\mathtt{F}_{\mathtt{T},\mathtt{F}/\mathtt{T}}(s,Z,false) :- \mathtt{F}_{\mathtt{T},\mathtt{F}}(s,Z)$$
$$\mathtt{F}_{\mathtt{T},\mathtt{T}}(s,Z) :- \mathtt{T}(s,Z)$$
$$\mathtt{F}_{\mathtt{T},\mathtt{F}}(s,Z) :- dom_{\mathtt{T.toCity}}(Z), \neg\, \mathtt{T}(s,Z)$$

Fig. 5: Example firing rules for WHYNOT $\mathtt{Q}(s,n)$

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 $\mathcal{PG}(P,I)$ 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, i.e., it may contain variables that only occur in negated goals. In fact, these variables should be restricted to the associated domains for the attributes the variables are bound to. Since the associated domain dom for an attribute R.A is given as an unary query $dom_{\mathtt{R.A}}$, we can use these queries directly in firing rules to restrict the values the variable is bound to. This is how we ensure that only missing answers formed from the associated domains are considered and that firing rules are always safe.

Example 7. *Reconsider the question* WHYNOT $\mathtt{Q}(s,n)$ *from Example 1. The firing rules generated for this question are shown in Fig. 5. We exclude the rules for the second goal* $\mathtt{T}(Z,n)$ *and the negated goal* $\neg\,\mathtt{T}(s,n)$ *which are analogous to the rules for the first goal* $\mathtt{T}(s,Z)$*. Tuple* $\mathtt{Q}(s,n)$ *is failed (not in the result), i.e., New York cannot be reached from Seattle with exactly one transfer. Hence, we are only interested in failed rule derivations of the rule* r_1 *with* $X{=}s$ *and* $Y{=}n$*. Each rule node in the provenance graph corresponding to such a rule derivation will only be connected to failed subgoals. Thus, we need to capture which goals are successful or failed for each such failed derivation. This can be modelled through boolean variables* V_1*,* V_2*, and* V_3 *(since there are three goals in the body) that are true if the corresponding goal is successful and false otherwise. The firing version* $\mathtt{F}_{\mathtt{r}_1,\mathtt{F}}(s,n,Z,V_1,V_2,\neg V_3)$ *of* r_1 *will contain all variable bindings for derivations of* r_1 *such that* $\mathtt{Q}(s,n)$ *is the head, the rule derivations are failed, and the* i^{th} *goal is successful (failed) for this binding iff* V_i *is true (false). Adding* $\mathtt{F}_{\mathtt{Q},\mathtt{F}}(s,n)$ *to the body of the firing rule ensures that* $\mathtt{Q}(s,n)$ *is not in* $\mathtt{Q}(I)$*. To produce all these bindings, we need rules capturing successful and failed tuple nodes for each subgoal of the rule* r_1*. We denote such rules using a* F/T *annotation and use a boolean variable (true or false) to record whether a tuple exists (e.g.,* $\mathtt{F}_{\mathtt{T},\mathtt{F}/\mathtt{T}}(s,Z,true) :- \mathtt{F}_{\mathtt{T},\mathtt{T}}(s,Z)$ *is one of these rules). Negated goals are dealt with by negating this boolean variable, i.e., the goal is successful if the corresponding tuple does not exist. For instance,* $\mathtt{F}_{\mathtt{T},\mathtt{F}/\mathtt{T}}(s,n,false)$ *represents the fact that tuple* $\mathtt{T}(s,n)$ *(a direct train connection from Seattle to New 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.,* $\mathtt{F}_{\mathtt{T},\mathtt{T}}(s,Z) :- \mathtt{T}(s,Z)$*), one for the failure case (e.g.,* $\mathtt{F}_{\mathtt{T},\mathtt{F}}(s,Z) :- dom_{\mathtt{T.toCity}}(Z), \neg\,\mathtt{T}(s,Z)$*), and two for the* F/T *firing version. For the failure case, we use predicate* $dom_{\mathtt{T.toCity}}$ *to only consider missing tuples* (s,Z) *where* Z *is a value from the associated domain of this attribute.*

Algorithm 1 Create Firing Rules

```
1:  procedure CreateFiringRules(P_A, Q(t))
2:      P_Fire ← []
3:      state ← typeof(Q(t))
4:      todo ← [Q(t)^state]
5:      done ← {}
6:      while todo ≠ [] do                    ▷ create rules for a predicate
7:          R(t)^σ ← pop(todo)
8:          insert(done, R(t)^σ)
9:          if isEDB(R) then
10:             CreateEDBFiringRule(P_Fire, R(t)^σ)
11:         else
12:             CreateIDBNegRule(P_Fire, R(t)^σ)
13:             rules ← getRules(R(t)^σ)
14:             for all r ∈ rules do          ▷ create firing rule for r
15:                 args ← (vars(r) − vars(head(r)))
16:                 args ← args(head(r)) :: args
17:                 CreateIDBPosRule(P_Fire, R(t)^σ, r, args)
18:                 CreateIDBFiringRule(P_Fire, R(t)^σ, r, args)
19:     return P_Fire
```

The algorithm that creates the firing rules for an annotated input query is shown as Algorithm 1 (the pseudocode for the subprocedures is in our technical report [21]). It maintains a list of annotated atoms that need to be processed which is initialized with the $Q(t)$ (PQ). For each such atom $R(t)^\sigma$ (here σ is the annotation of the atom), it creates firing rules for each rule r that has this atom as a head and a positive firing rule for $R(t)$. Furthermore, if the atom is annotated with F/T or F, then additional firing rules are added to capture missing tuples and failed rule derivations.

EDB atoms. For an EDB atom $R(t)^T$, we use procedure CREATEEDBFIRINGRULE to create one rule $\mathtt{F}_{\mathtt{R},\mathtt{T}}(t) :- R(t)$ 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 $dom(X_i)$ as explained in Example 7.

Rules. Consider a rule $r : R(t) :- g_1(\vec{X}_1), \ldots, g_n(\vec{X}_n)$. If the head of r is annotated with T, then we create a rule with head $\mathtt{F}_{r,\mathtt{T}}(\vec{X})$ where $\vec{X} = vars(r)$ 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 $\mathtt{F}_{r,\mathtt{F}}(\vec{X}, \vec{V})$ where $\vec{X}$ is defined as above, and $\vec{V}$ contains V_i if the i^{th} goal of r is positive and $\neg V_i$ otherwise. The body of this rule contains the F/T version of every goal in r's body plus an additional goal $\mathtt{F}_{\mathtt{R},\mathtt{F}}$ 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.

IDB atoms. For each rule r with head $R(t)$, we create a rule $\mathtt{F}_{\mathtt{R},\mathtt{T}}(t) :- \mathtt{F}_{r,\mathtt{T}}(\vec{X})$ where $\vec{X}$ 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 $\neg \mathtt{F}_{\mathtt{R},\mathtt{T}}(t)$ 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 m goals, and P_{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:*

- $\mathtt{F}_{\mathtt{R},\mathtt{T}}(t) \in P_{Fire}(I) \leftrightarrow R(t) \in P(I)$
- $\mathtt{F}_{\mathtt{R},\mathtt{F}}(t) \in P_{Fire}(I) \leftrightarrow R(t) \notin P(I)$
- $\mathtt{F}_{r,\mathtt{T}}(t) \in P_{Fire}(I) \leftrightarrow r(t) \models P(I)$
- $\mathtt{F}_{r,\mathtt{F}}(t, \vec{V}) \in P_{Fire}(I) \leftrightarrow r(t) \not\models P(I) \wedge head(r(t)) \notin P(I)$ *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 $d(R) = \max_{head(r)=R} d(r)$, i.e., the maximal depth among all rules r with $head(r) = R$; 3) the depth of a rule r is $d(r) = \max_{R \in body(r)} d(R) + 1$, 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: $d(P) = \max_{r \in P} d(r)$.

1) Base Case. Assume that we have a program P with depth 1, e.g., $r : \mathtt{Q}(\vec{X}) :- \mathtt{R}(\vec{X}_1), \ldots, \mathtt{R}(\vec{X}_n)$. We first prove that the positive and negative versions of firing rules for EDB atoms are correct, because only these rules are used for the rules of depth 1 programs. A positive version of EDB firing rule $\mathtt{F}_{\mathtt{R},\mathtt{T}}$ creates a copy of the input relation R and, thus, a tuple $t \in \mathtt{F}_{\mathtt{R},\mathtt{T}}$ iff $t \in R$. For the negative version $\mathtt{F}_{\mathtt{R},\mathtt{F}}$, all variables are bound to associated domains dom and it is explicitly checked that $\neg R(\vec{X})$ is true. Finally, $\mathtt{F}_{\mathtt{R},\mathtt{F}/\mathtt{T}}$ uses $\mathtt{F}_{\mathtt{R},\mathtt{T}}$ and $\mathtt{F}_{\mathtt{R},\mathtt{F}}$ (as third and fourth rules from the bottom in Fig. 5) to determine whether the tuple exists in R. Since these rules are correct, it follows that $\mathtt{F}_{\mathtt{R},\mathtt{F}/\mathtt{T}}$ is correct. The positive firing rule for the rule r ($\mathtt{F}_{r,\mathtt{T}}$) is correct since its body only contains positive and negative EDB firing rules ($\mathtt{F}_{\mathtt{R},\mathtt{T}}$ and $\mathtt{F}_{\mathtt{R},\mathtt{F}}$, respectively) which are already known to be correct. The correctness of the positive firing version of a rule's head predicate ($\mathtt{F}_{\mathtt{Q},\mathtt{T}}$) follows naturally from the correctness of $\mathtt{F}_{r,\mathtt{T}}$. The negative version of the rule $\mathtt{F}_{r,\mathtt{F}}(\vec{X}, \vec{V})$ contains an additional goal (i.e., $\neg \mathtt{Q}(\vec{X})$) and uses the firing version $\mathtt{F}_{\mathtt{R},\mathtt{F}/\mathtt{T}}$ to return only bindings for failed derivations. Since $\mathtt{F}_{\mathtt{R},\mathtt{F}/\mathtt{T}}$ has been proven to be correct, we only need to prove that the negative firing version of the head predicate of r is correct. For a head predicate with annotation F, we create two firing rules ($\mathtt{F}_{\mathtt{Q},\mathtt{T}}$ and $\mathtt{F}_{\mathtt{Q},\mathtt{F}}$). The rule $\mathtt{F}_{\mathtt{Q},\mathtt{T}}$ was already proven to be correct as in positive case. $\mathtt{F}_{\mathtt{Q},\mathtt{F}}$ is also correct, because it contains only $\mathtt{F}_{\mathtt{Q},\mathtt{T}}$ and domain queries in the body which were already known to be correct.

$$\begin{aligned}
\mathtt{F}_{\mathtt{Q},\mathtt{T}}(n,s) &:- \mathtt{F}_{r_1,\mathtt{T}}(n,s,Z)\\
\mathtt{F}_{r_1,\mathtt{T}}(n,s,Z) &:- \mathtt{F}_{\mathtt{T},\mathtt{T}}(n,Z), \mathtt{F}_{\mathtt{T},\mathtt{T}}(Z,s), \mathtt{F}_{\mathtt{T},\mathtt{F}}(n,s)\\
\mathtt{FC}_{r_2,r_1^1,\mathtt{T}}(n,Z) &:- \mathtt{T}(n,Z), \mathtt{F}_{r_1,\mathtt{T}}(n,s,Z)\\
\mathtt{FC}_{r_2,r_1^2,\mathtt{T}}(Z,s) &:- \mathtt{T}(Z,s), \mathtt{F}_{r_1,\mathtt{T}}(n,s,Z)\\
\mathtt{F}_{\mathtt{T},\mathtt{F}}(n,s) &:- \neg\, \mathtt{T}(n,s)
\end{aligned}$$

Fig. 6: Example firing rules with connectivity checks

2) Inductive Step. It remains to be shown that firing rules for programs of depth $n+1$ are correct. Assume that firing rules for programs of depth up to n are correct. Let r be a firing rule of depth $n+1$ in a program of depth $n+1$. It follows that $\max_{R \in body(r)} d(R) \leq n$ (i.e., the maximum depth among all predicates in the body of r should be n or less), otherwise r would be of a depth larger than $n+1$. Based on the induction hypothesis, it is guaranteed that the firing rules for all these predicates are correct. Using the same argument as in the base case, it follows that the firing rule for r is correct. ■

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.

Example 8. *Consider the firing rules for* WHY $\mathtt{Q}(n,s)$ *shown in Fig. 4. The corresponding rules with connectivity checks are shown in Fig. 6. All the rule nodes corresponding to $\mathtt{F}_{r_1,\mathtt{T}}(n,s,Z)$ are guaranteed to be connected to the PQ node $\mathtt{Q}(n,s)$. For sake of the example, assume that instead of using* T*, rule r_1 uses an IDB relation R which is computed using another rule $r_2 : \mathtt{R}(X,Y) :- \mathtt{T}(X,Y)$. Consider the firing rule $\mathtt{F}_{r_2,\mathtt{T}}(n,Z) :- \mathtt{T}(n,Z)$ created based on the first goal of r_1. Some provenance graph fragments computed by this rule may not be connected to $\mathtt{Q}(n,s)$. A tuple node $\mathtt{R}(n,c)$ for a constant c is only connected to the node $\mathtt{Q}(n,s)$ iff it is part of a*

Algorithm 2 Add Connectivity Joins

```
1: procedure AddConnectivityRules(P_Fire, Q(t))
2:     P_FC ← []
3:     paths ← PathStartingIn(P_Fire, Q(t))
4:     for all p ∈ paths do
5:         p ← FilterRuleNodes(p)
6:         for all e = (r_i(X⃗_1)^σ1, r_j(X⃗_2)^σ2) ∈ p do
7:             goals ← GetMatchingGoals(e)
8:             for all g_k ∈ goals do
9:                 g_new ← UnifyHead(F_{r_i,σ1}(t_1), g_k, F_{r_j,σ2}(t_2))
10:                r_new ← FC_{r_j,r_i^k,σ2}(t_2) :- body(F_{r_j,σ2}(t_2)), g_new
11:                P_FC ← P_FC :: r_new
12:    return P_FC
```

successful binding of r_1. That is, for the node $\mathtt{R}(n, c)$ to be connected, there has to exist another tuple (c, s) in $\mathtt{R}$. We check connectivity to $\mathtt{Q}(n, s)$ one hop at a time. This is achieved by adding the head of the firing rule for r_1 ($\mathtt{F}_{r_1,\mathtt{T}}(n, s, Z)$) to the body of the firing rule for r_2 as shown in Fig. 6 (the second and third rule from the bottom). We use $\mathtt{FC}_{r_2,r_1^k,\mathtt{T}}(\vec{X})$ to denote the firing rule for r_2 connected to the k^{th} goal of rule r_1. Note that, this connectivity check is unnecessary for rules with only constants (the last rule in Fig. 6).

Algorithm 2 traverses the query's rules starting from the PQ to find all combinations of rules r_i and 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 in the body of r_i, we create a rule $\mathtt{FC}_{r_j,r_i^k,\sigma_2}(\vec{X})$ 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 $\mathtt{T}(n, s)$ that is successful would be represented as $f_T^T(n, s)$. 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

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 $\mathtt{Q}(n, s)$ 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 $g_1^1(n, Z)$, and so on.*

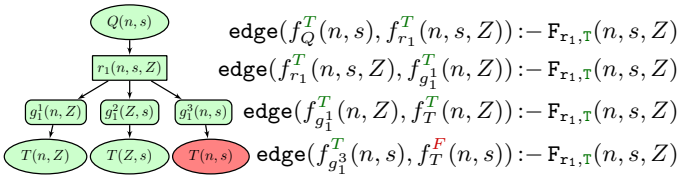


Fig. 7: Rules deriving the edge relation of an explanation

F. Correctness

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

Theorem 2 (Correctness)**.** *Datalog program $\mathbb{GP}(P, Q(t), I)$ generated for input program P, provenance question $Q(t)$, and instance I returns the edge relation of $\textsc{Expl}(P, Q(t), I)$.*

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

1. The constant values used as variable binding by the rules creating edges in $\mathbb{GP}(P, Q(t), I)$ 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 $\mathbb{GP}(P, Q(t), I)$ is strictly based on the structure of the input program and connects nodes that agree on variable bindings. Thus, each edge produced by $\mathbb{GP}(P, Q(t), I)$ will be contained in $\mathcal{PG}(P, I)$.

2. We now prove that the program $\mathbb{GP}(P, Q(t), I)$ returns precisely the set of edges of $\textsc{Expl}(P, Q(t), I)$. Assume that $Q(t)$ only contains constants (the extension to questions with 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

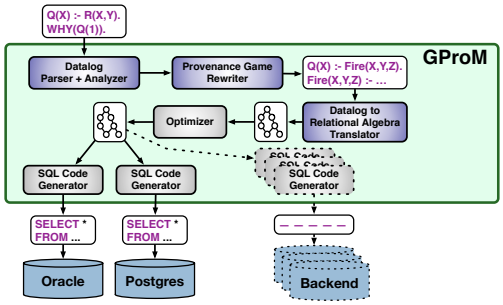


Fig. 8: Implementation in GProM

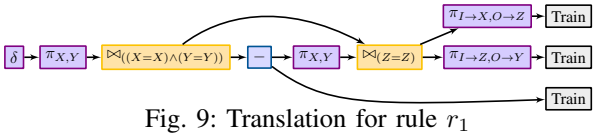


Fig. 9: Translation for rule r_1

$$r_1 : \texttt{only2hop}(X,Y) :- \texttt{DBLP}(X,Z), \texttt{DBLP}(Z,Y), \neg\, \texttt{DBLP}(X,Y)$$

$$r_2 : \texttt{XwithYnotZ}(X,Y) :- \texttt{DBLP}(X,Y), \neg\, \texttt{Q}_1(X)$$
$$r_{2'} : \texttt{Q}_1(X) :- \texttt{DBLP}(X, \text{`Svein Johannessen'})$$

$$r_3 : \texttt{only3hop}(X,Y) :- \texttt{DBLP}(X,A), \texttt{DBLP}(A,B), \texttt{DBLP}(B,Y), \neg\, \texttt{E}_1(X), \neg\, \texttt{E}_2(X)$$
$$r_{3'} : \texttt{E}_1(X) :- \texttt{DBLP}(X,Y)$$
$$r_{3''} : \texttt{E}_2(X) :- \texttt{DBLP}(X,A), \texttt{DBLP}(A,Y)$$

$$r_4 : \texttt{ordPriority}(X,Y) :- \texttt{CUSTOMER}(A,X,B,C,D,E,F,G), \texttt{ORDERS}(A,H,I,J,K,Y,M,N,O)$$

$$r_5 : \texttt{ordDisc}(X,Y) :- \texttt{CUSTOMER}(A,X,C,D,E,F,G,H), \texttt{ORDERS}(I,A,J,K,L,M,O,P,Q), \texttt{LINEITEM}(I,R,S,T,U,V,Y,W,Z,A',B',C',D',E',F',G')$$

$$r_6 : \texttt{partNotAsia}(X) :- \texttt{PART}(A,X,B,C,D,E,F,G,H), \texttt{PARTSUPP}(A,I,J,K,L), \texttt{SUPPLIER}(I,M,N,O,P,Q,R), \texttt{NATION}(O,S,T,U), \neg\, \texttt{R}_1(T, \text{`ASIA'})$$
$$r_{6'} : \texttt{R}_1(T,Z) :- \texttt{REGION}(T,Z,V)$$

Fig. 10: DBLP and TPC-H queries for experiments

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 an input. Our system parses and semantically analyzes this input. Schema information is gathered by querying the catalog of the backend database (e.g., to determine whether an EDB predicate with the expected arity exists). Modules for accessing schema information are already part of the GProM system, but a new semantic analysis component had to be developed to support Datalog. The algorithms presented in Sec. IV are applied to create the program $\mathbb{GP}(P, Q(t), I)$ which computes $\textsc{Expl}(P, Q(t), I)$. This program is translated into a relational algebra ($\mathcal{RA}$). The resulting algebra expression is then translated into SQL and sent to the backend database to compute the edge relation of the explanation for the PQ. Based on this edge relation, we then render a provenance graph (e.g., the graphs shown in Fig. 2 and 3 are actual results produced by the system).[1] While it would certainly be possible to directly translate the Datalog program into SQL without the intermediate translation into $\mathcal{RA}$, we choose to introduce this step to be able to leverage the existing heuristic and cost-based optimizations for $\mathcal{RA}$ graphs built into GProM and use its library of $\mathcal{RA}$ to SQL translators.

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 an expression for each IDB predicate as the union of the expressions for all rules with the predicates in the head. Finally, the algebra expressions for 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.*

[1] More examples for our method and installation guideline for GProM are available at https://github.com/IITDBGroup/gprom/wiki/datalog_prov.

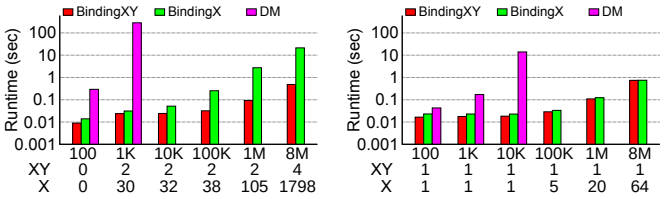
VI. Experiments

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). To the best of our knowledge, there are no openly available implementations for missing answers that we could compare against. Thus, 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.

Workloads. We compute explanations for the queries in Fig. 10 over the datasets we have introduced. For the DBLP dataset, we consider: `only2hop` (r_1) which is our running example query; `XwithYnotZ` (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

Num of Vars \ DBLP (#tuples)	100	1K	10K	100K
2 Variables (r_2)	0.043	0.171	14.016	-
3 Variables (r_1)	0.294	285.524	-	-
4 Variables (r_3)	56.070	-	-	-
Num of Vars \ TPC-H (Size)	**10MB**	**100MB**	**1GB**	**10GB**
($>$ 10) Variables (r_4, r_5, r_6)	-	-	-	-

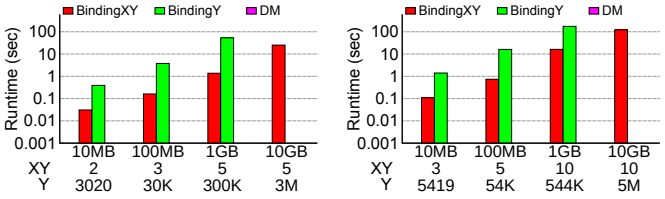
Fig. 11: Runtime of DM in seconds. For entries with '-', the computation did not finish in the allocated time of 10 min.



(a) Runtime of only2hop (b) Runtime of XwithYnotZ

Query \ Binding	X	Y
(a) only2hop	Tore Risch	Rafi Ahmed
(b) XwithYnotZ	Arjan Durresi	Raj Jain

(c) Variable bindings for DBLP PQs



(d) Runtime of ordPriority (e) Runtime of ordDisc

Query \ Binding	X	Y
(d) ordPriority	Customer16	1-URGENT
(e) ordDisc	Customer16	0

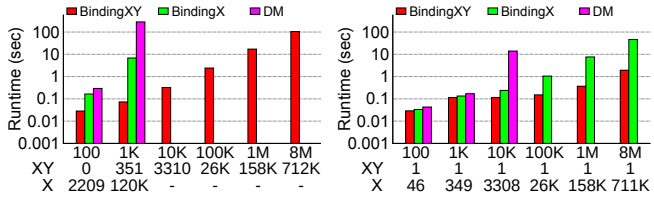
(f) Variable bindings for TPC-H PQs

Fig. 12: Runtime - Why questions

(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.

Implementing DM. As introduced in Sec. I, DM has to instantiate a graph with $\mathcal{O}(\|adom(I)\|^n)$ 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



(a) Runtime of only2hop (b) Runtime of XwithYnotZ

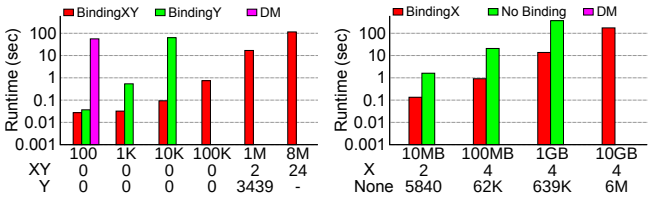
Query \ Binding	X	Y
(a) only2hop	Tore Risch	Svein Johannessen
(b) XwithYnotZ	Tor Skeie	Joo-Ho Lee

(c) Variable bindings for DBLP PQs

Fig. 13: Runtime - Why-not questions

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 (BindingX), then the queries determine authors that occur together with the author 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 (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 (BindingY) in queries r_4 and r_5 expresses a condition, e.g., Y = '1-URGENT' in r_4 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 DBLP 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 co-author of "Svein Johannessen". If one variable is bound (BindingX), then the why-not question explains for pairs of authors where one of the authors is bound to X, why the pair does not appear together in the query result. The results for queries on r_1 and r_2 are shown in Fig. 13.a and 13.b., respectively. The number of output tuples produced by the provenance computation (we show the number of generated rule nodes below the X axis) is quadratic in the database size resulting in a quadratic increase in runtime. Our approach significantly improves the performance compared to DM which is limited to very small datasets (less than 10K). Limiting the result size of missing answer questions for queries with many existential variables (r_4, r_5, and r_6) would require aggressive summarization techniques, which we leave for future work.



(a) Runtime of `only3hop` (b) Runtime of `partNotAsia`

Query \ Binding	X	Y
(a) `only3hop`	Alex Benton	Paul Erdoes
(b) `partNotAsia`	grcpi[1]	-

[1] grcpi = ghost royal chocolate peach ivory

(c) Variable bindings for DBLP and TPC-H PQs

Fig. 14: Runtime - Why questions over queries with negation

Queries with Negation. In this experiment, we measure performance of our approach for why questions over queries with negation. We choose rules r_3 (multiple negated goals) and r_6 (one negated goal) shown in Fig. 10. We use the bindings shown in Fig. 14.c. The results shown in Fig. 14.a and 14.b demonstrate that our approach efficiently computes explanations for r_3 and r_6, respectively. When increasing the database size, the runtimes of PQs for these queries exhibit the same trend as observed for other why (why-not) questions and significantly outperform DM. For instance, the performance of the query `partNotAsia` (Fig. 14.b), which contains many variables and negation, exhibits the same trend as queries that have no negation (i.e., r_4 and r_5).

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. A drawback of our current approach is the potentially large size of explanations for missing answers (exponential in the number of existential variables for rules deriving missing answers). In future work, we will investigate how to summarize and generalize provenance (e.g., in the spirit of [5], [10], [12], [25]) to address this problem. Other topics of interest include considering integrity constraints in the provenance graph construction (e.g., derivations cannot succeed if they violate 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).

Acknowledgments. Work supported in part by NSF awards SMA-1637155 and ACI-1541450.

References

[1] B. Arab, D. Gawlick, V. Radhakrishnan, H. Guo, and B. Glavic. A generic provenance middleware for database queries, updates, and transactions. In *TaPP*, 2014.

[2] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. In *PODS*.

[3] N. Bidoit, M. Herschel, and K. Tzompanaki. Immutably answering why-not questions for equivalent conjunctive queries. In *TaPP*, 2014.

[4] N. Bidoit, M. Herschel, K. Tzompanaki, et al. Query-Based Why-Not Provenance with NedExplain. In *EDBT*, pages 145–156, 2014.

[5] B. t. Cate, C. Civili, E. Sherkhonov, and W.-C. Tan. High-level why-not explanations using ontologies. In *PODS*, pages 31–43, 2014.

[6] A. Chapman and H. V. Jagadish. Why Not? In *SIGMOD*, pages 523–534, 2009.

[7] J. Cheney, L. Chiticariu, and W. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[8] D. Deutch, A. Gilad, and Y. Moskovitch. Selective provenance for datalog programs using top-k queries. *PVLDB*, 8(12):1394–1405, 2015.

[9] D. Deutch, T. Milo, S. Roy, and V. Tannen. Circuits for datalog provenance. In *ICDT*, pages 201–212, 2014.

[10] K. El Gebaly, P. Agrawal, L. Golab, F. Korn, and D. Srivastava. Interpretable and informative explanations of outcomes. *PVLDB*, 8(1):61–72, 2014.

[11] J. Flum, M. Kubierschky, and B. Ludäscher. Total and partial well-founded datalog coincide. In *ICDT*, pages 113–124, 1997.

[12] B. Glavic, S. Köhler, S. Riddle, and B. Ludäscher. Towards constraint-based explanations for answers and non-answers. In *TaPP*, 2015.

[13] B. Glavic, R. J. Miller, and G. Alonso. Using sql for efficient generation and querying of provenance information. In *In search of elegance in the theory and practice of computation*, pages 291–320. 2013.

[14] T. J. Green, M. Aref, and G. Karvounarakis. Logicblox, platform and language: A tutorial. In *Datalog in Academia and Industry*, pages 1–8. Springer, 2012.

[15] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update Exchange with Mappings and Provenance. In *VLDB*, pages 675–686, 2007.

[16] M. Herschel and M. Hernandez. Explaining Missing Answers to SPJUA Queries. *PVLDB*, 3(1):185–196, 2010.

[17] J. Huang, T. Chen, A. Doan, and J. Naughton. On the provenance of non-answers to queries over extracted data. In *VLDB*, pages 736–747, 2008.

[18] G. Karvounarakis and T. J. Green. Semiring-annotated data: queries and provenance. *SIGMOD Record*, 41(3):5–14, 2012.

[19] S. Köhler, B. Ludäscher, and Y. Smaragdakis. Declarative datalog debugging for mere mortals. In *Datalog 2.0: Datalog in Academia and Industry*, pages 111–122, 2012.

[20] S. Köhler, B. Ludäscher, and D. Zinn. First-order provenance games. In *In Search of Elegance in the Theory and Practice of Computation*, pages 382–399. 2013.

[21] S. Lee, S. Köhler, B. Ludäscher, and B. Glavic. Efficiently computing provenance graphs for queries with negation. Technical Report CoRR, arXiv:1701.05699, 2016.

[22] S. Lee, S. Köhler, B. Ludäscher, and B. Glavic. Implementing Unified Why- and Why-Not Provenance Through Games. In *TaPP (Poster)*, 2016.

[23] A. Meliou, W. Gatterbauer, K. Moore, and D. Suciu. The Complexity of Causality and Responsibility for Query Answers and non-Answers. *PVLDB*, 4(1):34–45, 2010.

[24] E. Pontelli, T. C. Son, and O. Elkhatib. Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming*, 9(01):1–56, 2009.

[25] S. Riddle, S. Köhler, and B. Ludäscher. Towards constraint provenance games. In *TaPP*, 2014.

[26] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *SIGMOD*, pages 15–26, 2010.

[27] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems with negative provenance. In *SIGCOMM*, pages 383–394, 2014.

[28] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at internet-scale. In *SIGMOD*, pages 615–626, 2010.