PUG: A Framework and Practical Implementation for Why & Why-Not Provenance (extended version)

Seokki Lee, Bertram Ludäscher, Boris Glavic

IIT DB Group Technical Report IIT/CS-DB-2018-02

2018-08

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

Noname manuscript No.
(will be inserted by the editor)

PUG: A Framework and Practical Implementation for Why & Why-Not Provenance (Extended version)

Seokki Lee · Bertram Ludäscher · Boris Glavic

Received: date / Accepted: date

Abstract Explaining why an answer is (or is not) returned by a query is important for many applications including auditing, debugging data and queries, and answering hypothetical questions about data. In this work, we present the first *practical* approach for answering such questions for queries with negation (first-order queries). Specifically, we introduce a graph-based provenance model that, while syntactic in nature, supports reverse reasoning and is proven to encode a wide range of provenance models from the literature. The implementation of this model in our PUG (Provenance Unification through Graphs) system takes a provenance question and Datalog query as an input and generates a Datalog program that computes an *explanation*, i.e., the part of the provenance that is relevant to answer the question. Furthermore, we demonstrate how a desirable factorization of provenance can be achieved by rewriting an input query. We experimentally evaluate our approach demonstrating its efficiency.

1 Introduction

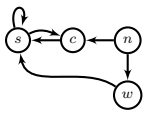
Provenance for relational queries records how results of a query depend on the query's inputs. This type of information can be used to explain *why* (and *how*) a result is derived by a query over a given database. Recently, provenance-like techniques have been used to explain why a tuple (or a set of tuples described declaratively by a pattern) is *missing* from the query result (see [19] for a survey covering both provenance and missing answer techniques). However, the two problems have been treated mostly in isolation. Consider the following observation in earlier work [24]: asking why a tuple t is absent from the result of a query Q is equivalent to asking why t is present in $\neg Q$ (i.e., the complement of the result of Q wrt. the active domain). Thus, a unification of *why* and *why-not* provenance is naturally achieved by developing a provenance model for queries with negation. The approach for provenance and missing answers from [41] is based on the same observation.

In this paper, we introduce a graph model for provenance of first-order (FO) queries expressed as *non-recursive Datalog queries with negation*[1] (or *Datalog* for short) and an efficient method for explaining a (missing) answer using SQL. Our approach is based on the observation that typically only a part of the 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	$\mathbb{N}[X]$
seattle	seattle	p
seattle	chicago	q
chicago	seattle	r
washington dc	seattle	s
new york	washington dc	t
new york	chicago	u



Result of query **Q**

X	Y
washington dc	chicago
new york	seattle
chicago	chicago

Fig. 1: Example train connection database and query

✉ Seokki Lee, Boris Glavic
10 W 31st Street, Chicago, IL 60616
E-mail: slee195@hawk.iit.edu, E-mail: bglavic@iit.edu

Bertram Ludäscher
501 E. Daniel St, Champaign, IL 61820
E-mail: ludaesch@illinois.edu

[1] or, equivalently, queries in full relational algebra (without aggregation), formulas in FO logic under the closed world assumption, and SPJUD-queries (select, project, join, union, difference).

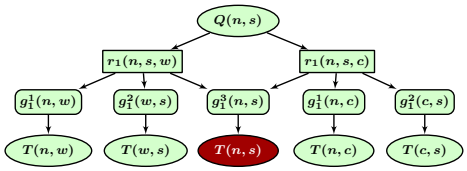


Fig. 2: Provenance graph explaining WHY $\mathtt{Q}(n, s)$

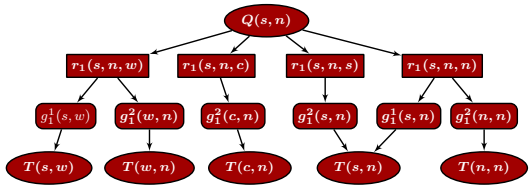


Fig. 3: Provenance graph explaning WHYNOT $\mathtt{Q}(s, n)$

Example 1. *Consider the relation* `Train` *in Fig. 1 that stores train connections. 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. These graphs contain three types of 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 in *light green* (successful/existing) or *dark 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 stopping in Washington DC or Chicago and there is no direct connection between these two cities. These 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 *representing these 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 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. For that, we have to decide how to bound the set of missing answers to be considered. 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, we refer to this set of values as the active domain $adom(I)$ of a database instance I.

Example 3. *Fig. 3 shows the explanation for* WHYNOT $\mathtt{Q}(s, n)$*, 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 result because all potential ways of deriving this tuple through 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. 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 (*tuple node $\mathtt{T}(s, w)$ *in red), and Washington DC to New York (*tuple node $\mathtt{T}(w, n)$ *in red). 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.*

Observe that nodes for missing tuples and successful rule derivations are conjunctive in nature (they depend on all their children) while existing tuples and failed rule derivations are disjunctive (they only require at least one of their children to be present).

Provenance Model. By recording which rule derivations justify the existence or absence of a query result, our model is suited well for debugging both data and queries. However, simpler provenance types, e.g., only tracking data dependencies, are sufficient for some applications. For example, assume that we record as for each train connection from which webpage we retrieved information about this train connection. A user may be interested in knowing based on which webpages a query answer was computed. This question can be answered using a simpler provenance type called Lineage

(semiring $\mathsf{Which}(X)$ [18]) which records the set of input tuples a result depends on. For such applications, we prefer simpler provenance types, because they are easier to interpret and more efficient to compute. Importantly, only minor modifications to our framework were required to support such provenance types.

Relationship to Other Provenance Models. In comparison to other provenance models, our model is more syntax-driven. We argue that it is actually a feature (not a bug). An important question is what is the semantic justification of our model, i.e., how do we know whether it correctly models Datalog query evaluation semantics and whether all (and only) relevant provenance is captured. First, we observe that our model encodes Datalog query semantics by construction. We justify that all relevant provenance is captured indirectly by demonstrating that our model captures sufficient information to derive provenance according to well-established models. Specifically, we demonstrate that our model is equivalent to provenance games [24] which also support FO queries. It was proven in [24] that provenance polynomials, the most general form of provenance in the semiring model [18,22], for a result of a positive query can be "read out" from a provenance game. By being equivalent to provenance games, our provenance model also enjoys this property. We extend this result to queries with negation by relating our model to semiring provenance for FO model checking [13,37,41]. We prove that, for any FO formula φ, we can generate a query Q_φ such that the semiring provenance annotation of the formula $\pi(\varphi)$ according to a $\mathcal{K}$-interpretation π (annotation of positive and negated literals [13]) can be extracted efficiently from our provenance graph for Q_φ. Note that non-recursive Datalog queries with negation and FO formulas under the closed world assumption have the same expressive power and, thus, we use these languages interchangeably.

Reverse Reasoning (How-to Queries). For some applications, a user may not be interested in how a result was derived, but wants to understand how a result of interest can be achieved through updates to the database (see e.g., [28,37,29]). We extend our approach to support such "reverse reasoning" by introducing a third possible state of nodes in a provenance graph reserved for facts and derivations whose truth is undetermined. The provenance graph generated over an instance with undetermined facts represents a set of provenance graphs - one for each instance that is derived by assigning a truth value to each undetermined fact. We demonstrate that these graphs can be used to compute the semiring provenance of an FO formula under a provenance tracking interpretation as defined in [37].

Computing Explanations. We utilize Datalog to generate provenance graphs that explain a (missing) query result. Specifically, we instrument the input Datalog program to compute provenance bottom-up. Evaluated over an instance I, the instrumented program returns the edge relation of an explanation (provenance graph).

The main driver of our approach is a rewriting of Datalog rules (so-called *firing rules*) that captures successful and failed rule derivations. Firing rules for positive queries were first introduced in [23]. We have generalized this concept to negation and failed rule derivations. Firing rules provide sufficient information for constructing any of the provenance graph types we support. To make provenance capture efficient, we avoid capturing derivations that will not contribute to an explanation. We achieve this by propagating information from the user's provenance question throughout the query to prune derivations that 1) do not agree with the constants of the question or 2) cannot be part of the explanation based on their success/failure status. For instance, in the running example, $\mathtt{Q}(n, s)$ is only connected to derivations of rule r_1 with $X{=}n$ and $Y{=}s$.

We implemented our approach in *PUG* [25] (Provenance Unification through Graphs), an extension of our *GProM* [1] system. Using PUG, we compile rewritten Datalog programs into relational algebra, and translate such algebra expressions into SQL code that can be executed by a standard relational database backend.

Factorizing Provenance. Nodes in our provenance graphs are uniquely identified by their label. Thus, common subexpressions are shared leading to more compact graphs. For instance, observe that $g_1^3(n, s)$ in Fig. 2 is shared by two rule nodes. We exploit this fact by rewriting the input program to generate more concise, but equivalent, provenance. This is akin to factorization of provenance polynomials in the semiring model and utilizes factorized databases techniques [31,32].

Contributions. This paper extends our previous work [25] in the following ways: we extend our model to support less informative, but more concise, provenance types; we extend our provenance model to support reverse reasoning [13] where the truth of some facts in the database is left undetermined; we demonstrate that our provenance graphs (explanations) are equivalent to provenance games [24] and how semiring provenance and its FO extension as presented in [37] can be extracted from our provenance model; we demonstrate how to rewrite an input program to generate a desirable (concise) factorization of provenance and evaluate the performance impact of this technique; finally, we present an experimental comparison with the language-integrated provenance techniques implemented in Links [9].

The remainder of this paper is organized as follows. We discuss related work in Sec. 2 and review Datalog in Sec. 3. We define our model in Sec. 4 and prove it to be equivalent to provenance games [24] in Sec. 5. We, then, show how our approach relates to semiring provenance for positive queries and FO model checking in Sec. 6 and 7, respectively. We present our approach for computing explanations in Sec. 8, and factorization in Sec. 9. Sec. 10 covers our implementation in *PUG* which we evaluate in Sec. 11. We conclude in Sec. 12.

2 Related Work

Our provenance graphs have strong connections to other provenance models for relational queries and to approaches for explaining missing answers.

Provenance Games. Provenance games [24] 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. By virtue of supporting negation, provenance games can uniformly answer why and why-not questions. We prove equivalence of our approach with provenance games in Sec. 5. Köhler et al. [24] present an algorithm that computes the provenance game for a program P and database 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 [10]. In constrast, our approach directly computes succinct explanations that contain only relevant provenance.

Database Provenance. Several provenance models for database queries have been introduced in related work, e.g., see [5,22]. The semiring annotation framework generalizes these models for positive relational algebra (and, thus, positive non-recursive Datalog). An essential property of the $\mathcal{K}$-relational model is that the semiring of *provenance polynomials* $\mathbb{N}[X]$ generalizes all other semirings. It has been shown in [24] that provenance games generalize $\mathbb{N}[X]$ for positive queries. Since our graphs are equivalent to provenance games in the sense that there exist lossless transformations between both models (see Sec. 5), our graphs also encode $\mathbb{N}[X]$ and, thus, all other provenance models expressible as semirings (see Sec. 6.2). Provenance graphs which are similar to our graphs restricted to positive queries have been used as graph representations of semiring provenance (e.g., see [7,8,22]). Both our graphs and the boolean circuits representation of semiring provenance [8] explicitly share common subexpressions in the provenance. While these circuits support recursive queries, they do not support negation. Recently, extension of circuits for semirings with monus (supporting set difference) have been discussed [36]. The semiring model has also been applied to record provenance of model checking for first-order (FO) logic formulas [37,13,41]. This work also supports missing answers using the observation made earlier in [24]. Support for negation relies on 1) translating formulas into negation normal form (*nnf*), i.e., pushing all negations down to literals, and 2) annotating both positive and negative literals using a separate set X and $\bar{X}$ of indeterminates in provenance expressions where variables from X are reserved for positive literals and variables from $\bar{X}$ for negated literals. This idea of using dual (positive and negative) indeterminates is an independent rediscovery of the approach from [6] which applied this idea for FO queries. The main differences between these approaches are 1) that the results from [6] where only shown for one particular semiring ($Bool(X \cup \bar{X})$, the semiring of boolean expressions over dual indeterminates) and 2) that [6] supports recursion in the form of well-founded Datalog and answer set programming (disjunctive Datalog). We prove that our model encompasses the model from [13]. The notion of causality is also closely related to provenance. Meliou et al. [27] compute causes for answers and non-answers. However, the approach requires the user to specify which missing inputs are considered as causes for a missing output. Roy et al. [34,35] employ causality to compute explanations for high or low outcomes of aggregation queries as sets of input tuples which have a large impact on the result. Such sets of tuples are represented compactly through selection queries. A similar method was developed independently by Wu et al. [39].

Why-not and Missing Answers. Approaches for explaining missing answers are either based on the query [2, 3,4,38] (i.e., which operators filter out tuples that would have contributed to the missing answer) or based on the instance [20,21] (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. [4]. Huang et al. [21] first introduced an instance-based approach. Since then, several techniques have been developed to exclude spurious explanations, to support larger classes of queries [20], and to support distributed Datalog systems in Y! [40]. 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 insertion has to be translated as an insertion 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 [11, 33] and provenance graphs [26] that are particularly useful for negation and missing answers to deal with the potentially large size of the resulting provenance. Similarly, some missing answer approaches [20] 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 derivations was introduced by Köhler et al. [23]. They apply this idea for provenance-based debugging of positive Datalog. Firing rules are also similar to relational implementations of provenance capture in Perm [12], LogicBlox [16], Orchestra [17], and GProM [1]. Zhou et al. [42] leverage such rules for the distributed ExSPAN system using either full propagation or reference based provenance. The extension of firing rules for negation is the main enabler of our approach.

3 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 as $head(r_i)$, 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 (FO queries), so goals $\mathtt{R_j}(\vec{X_j})$ in the body are *literals*, i.e., atoms $\mathtt{L}(\vec{X_j})$ or their negation $\neg\mathtt{L}(\vec{X_j})$, and 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 ($\{X, Y\}$) and the variables in the negated goal ($\{X, Y\}$) also occur positively in the body. 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.

4 Provenance Model

We now introduce our provenance model and formalize the problem addressed in this work: compute 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.

4.1 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. 1, we have to decide how to bound the set of missing answers. 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. 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 $\mathtt{R.A}$. These associated

domains fulfill two purposes: 1) to reduce the size of explanations and 2) to avoid semantically meaningless answers. For instance, if there exists 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 as 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.

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. In Fig. 1, $attrs(r_1, Z)$=$\{$`Train.fromCity`, `Train.toCity`$\}$. We say a rule derivation $r(c_1, \ldots, c_n)$ is *domain grounded* iff $c_i \in \bigcap_{A \in attrs(r,X_i)} dom(A)$ for all $i \in \{1, \ldots, n\}$. For a relation $\mathtt{R}(\mathtt{A_1}, \ldots, \mathtt{A_n})$, we use $\textsc{Tup}(\mathtt{R})$ to denote the set of all possible tuples for `R`, i.e., $\textsc{Tup}(\mathtt{R}) = dom(\mathtt{R.A_1}) \times \ldots \times dom(\mathtt{R.A_n})$.

4.2 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. Note that the first type of labels uniquely identifies nodes.

Definition 2 (Provenance Graph)**.** *Let P be a Datalog program, I a database instance, dom a domain assignment for I, and $\mathbb{L}$ the domain of 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 $\mathcal{S} : V \to \{T, F\}$ (T for true/success and F for false/failure). We require that $\forall v, v' \in V : \mathcal{L}(v) = \mathcal{L}(v') \to v = v'$. The graph $\mathcal{PG}(P, I)$ 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 $\mathtt{Q}(n, s) \in P(I)$ (Fig. 2), this tuple node is connected to all successful derivations with $\mathtt{Q}(n, s)$ in the head which in turn are connected to goal nodes for each of the three goals of rule r_1. $\mathtt{Q}(s, n) \notin P(I)$ (Fig. 3) and, thus, its node is connected to all failed derivations with $\mathtt{Q}(s, n)$ as a head. Here, we have assumed that all cities can be considered as start and end points of missing train connections, i.e., both $dom(\mathtt{T.fromCity})$ and $dom(\mathtt{T.toCity})$ are defined as $adom(\mathtt{T.fromCity}) \cup adom(\mathtt{T.toCity})$. Thus, we have considered derivations $r_1(s, n, Z)$ for $Z \in \{c, n, s, w\}$.

4.3 Provenance 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 specified as a pattern and paired with a qualifier (WHY / WHYNOT) 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 WHYNOT $\mathtt{Q}(n, X)$ we return all explanations for missing tuples where the first attribute is n, i.e., why it is not the case that 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, Q an IDB predicate, and dom a domain assignment for I. A* provenance question ψ *is of the form* $\text{WHY}\,Q(t)$ *or* $\text{WHYNOT}\,Q(t)$ *where $t = (v_1, \ldots, v_n)$ consists of variables and domain constants ($dom(\texttt{Q.A})$ for each attribute $\texttt{Q.A}$). We define:*

$$\begin{aligned}\textsc{Pattern}(\psi) &= Q(t)\\ \textsc{Match}(\text{WHY}\,Q(t)) &= \{Q(t') | t' \in P(I) \wedge t' \preccurlyeq t\}\\ \textsc{Match}(\text{WHYNOT}\,Q(t)) &= \{Q(t') | t' \notin P(I) \wedge t' \preccurlyeq t \wedge t' \in \textsc{Tup}(Q)\}\end{aligned}$$

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 these PQs. We now formally define this type of explanation.

Definition 4 (Explanation)**.** *The* explanation $\textsc{Expl}(P, \psi, I, dom)$ *for a* PQ ψ *according to P, I, and dom is the subgraph of $\mathcal{PG}(P, I)$ containing only nodes that are connected to at least one node in* $\textsc{Match}(\psi)$.

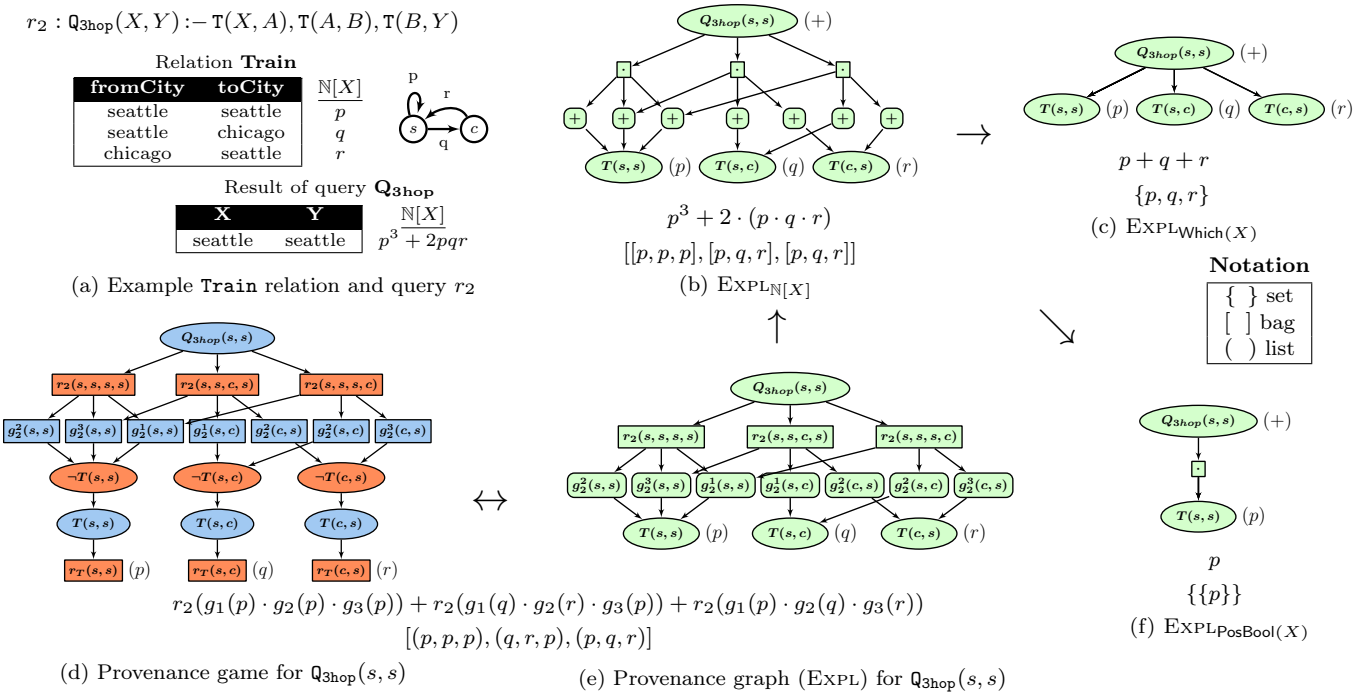
In the following we will drop *dom* from $\textsc{Expl}(P, \psi, I, dom)$ if it is clear from the context or irrelevant for the discussion. Given this definition of explanation, note that 1) all nodes connected to a tuple node matching the PQ are relevant for computing this tuple and 2) only nodes connected to this node are relevant for the outcome. Consider $Q(t') \in \textsc{Match}(\psi)$ for a question $\text{WHY}\,Q(t)$. Since $Q(t') \in P(I)$, all successful derivations with head $Q(t')$ justify the existence of t' and these are precisely the rule nodes connected to $Q(t')$ in $\mathcal{PG}(P, I)$. For $\text{WHYNOT}\,Q(t)$ and matching $Q(t')$ we have $Q(t') \notin P(I)$ which is the case if all derivations with head $Q(t')$ have failed. In this case, all such derivations 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.

5 Provenance Graphs and Provenance Games

We now prove that provenance graphs according to Def. 2 are equivalent to provenance games. Thus, our model inherits the semantic foundation of provenance games. Specifically, provenance games were shown to encode Datalog query evaluation. Furthermore, the interpretation of provenance game graphs as 2-player games provides a strong justification for why the nodes reachable from a tuple node justify the existence/absence of the tuple. We show how to transform a provenance game $\Gamma(P, \psi, I)$ into an explanation $\textsc{Expl}(P, \psi, I)$ and vice versa to demonstrate that both are equivalent representations of provenance. We define a function $\textsc{Tr}_{\Gamma \to \textsc{Expl}}$ that maps provenance games to graphs and its inverse $\textsc{Tr}_{\textsc{Expl} \to \Gamma}$. Before that, we first give an overview of provenance games (see [24] for more details).

Provenance Games. Similar to our provenance graphs, provenance games are graphs that record successful and failed rule derivations. Provenance games consist of four types of nodes (e.g. Fig. 4d): rule nodes (boxes labeled with a rule identifier and the constant arguments of a rule derivation), goal nodes (boxes labeled with a rule identifier and the goal's position in the rule's body), tuple nodes (ovals), and EDB fact nodes (boxes labeled with an EDB relation name and the constants of a tuple). Every tuple node in a provenance game appears both positively and negatively, i.e., for every tuple node $R(t)$, there exists a tuple node $\neg R(t)$. Given a program P and database instance I, a provenance game is constructed by creating a positive and negative tuple node $\texttt{R}(c_1, \cdots, c_n)$ for each n-ary predicate $\texttt{R}$ and for all combinations of constants c_i from the active domain $adom(I)$. Similarly, nodes are created for rule derivations, i.e., a rule where variables have been replaced with constants from $adom(I)$ and each goal in the body of a rule (similar to Def. 2). In the game, a derivation of rule r for a vector of constants $\vec{c}$ is labeled as $r(\vec{c})$, e.g., a derivation $\texttt{Q}_{\texttt{3hop}}(s, s) :- \texttt{T}(s, c), \texttt{T}(c, s), \texttt{T}(s, s)$ of r_2 in Fig. 4a is represented as a rule node labeled with $r_2(s, s, c, s)$. Finally, EDB fact nodes are added for each tuple in I, e.g., $r_T(s, s)$ for the tuple (seattle, seattle) in the `Train` relation (Fig. 4a). Tuple nodes are connected to the grounded rule nodes that derive them (have the tuple in their head), rule nodes to goal nodes for the grounded goals in their body, and goal nodes to negated tuple nodes corresponding to the goal (positive goals) or positive tuple nodes (negated goals). Such a game is interpreted as a 2-player game where the players argue for/against the existence of a tuple in the result of evaluating P over I. The existence of strategies for a player in this game determines tuple existence and success of rule derivations. A *solved game* is one where each node in the game graph is labeled as either won W (there exists a strategy for the player starting in this position) or lost L (no such strategy exists). A tuple node $\texttt{R}(t)$ is labeled as W iff the tuple $\texttt{R}(t)$ exists. A corollary of this is that a rule is labelled L if the corresponding derivation is successful and W otherwise.[2] Given such a solved

[2] This follows from the semantics of the type of 2-player game used here. The details are beyond the scope of this paper.



(a) Example Train relation and query r_2 (b) $\text{Expl}_{\mathbb{N}[X]}$ (c) $\text{Expl}_{\text{Which}(X)}$

(d) Provenance game for $\mathtt{Q_{3hop}}(s,s)$ (e) Provenance graph (Expl) for $\mathtt{Q_{3hop}}(s,s)$ (f) $\text{Expl}_{\text{PosBool}(X)}$

Fig. 4: Transformations exemplified using the provenance graph for $\mathtt{Q_{3hop}}(s,s)$. For each graph, we show the structure of the provenance encoded by this graph and the corresponding semiring annotation where applicable.

game (denoted as $\Gamma(P,I)$), we can extract a subgraph rooted at an IDB tuple $\mathtt{Q}(t)$ as the provenance of $\mathtt{Q}(t)$. Similar to how we derive an explanation for a PQ with $\text{Pattern}(\psi) = Q(t)$ where t may contain variables as the subgraph of the provenance graph $\mathcal{PG}(P,I)$ containing all IDB tuple nodes matching t and nodes reachable from these nodes, we can derive the corresponding subgraph in the provenance game $\Gamma(P,I)$ and denote it as $\Gamma(P,\psi,I)$ (we call such subgraphs *game explanations*).

Translating between Provenance Graphs and Provenance Games. The translation $\text{Tr}_{\text{Expl}\to\Gamma}$ of a provenance graph into the corresponding game and the reverse transformation $\text{Tr}_{\Gamma\to\text{Expl}}$ are straight-forward. Thus, we only sketch $\text{Tr}_{\text{Expl}\to\Gamma}$ here. EDB tuple nodes are expanded to subgraphs $\neg\mathtt{R}(t) \to \mathtt{R}(t) \to r_R(t)$ for existing tuples and $\neg\mathtt{R}(t) \to \mathtt{R}(t)$ for missing tuples. IDB tuple nodes are always expanded to subgraphs of the later form. Rule and goal nodes and their interconnections are preserved. Goal nodes are connected to negated tuple nodes (positive goals) and to positive tuple nodes (negated goals). For positive tuple and goal nodes, we translate T to W (won) and F to L (lost). For negated tuple nodes and rule nodes, this mapping is reversed, i.e., T to L and F to W.

$\text{Tr}_{\text{Expl}\to\Gamma}$. $\text{Tr}_{\text{Expl}\to\Gamma}$ consists of the following steps executed in the order shown below:

- Replace each EDB tuple node v with $\mathcal{L}(v) = \mathtt{R}(t)$ and $\mathcal{S}(v) = T$ with a subgraph $v_1 \to v_2 \to v_3$ where $v_1 = \neg\mathtt{R}(t)$, $v_2 = \mathtt{R}(t)$, and $v_3 = r_R(t)$. Label these nodes as follows: $\lambda(v_1) = \lambda(v_3) = L$ and $\lambda(v_2) = W$.
- Replace each EDB tuple node v with $\mathcal{L}(v) = \mathtt{R}(t)$ and $\mathcal{S}(v) = F$ with a subgraph $v_1 \to v_2$ where $v_1 = \neg\mathtt{R}(t)$ and $v_2 = \mathtt{R}(t)$ using labels $\lambda(v_1) = W$ and $\lambda(v_2) = L$.
- Replace each IDB tuple node v with $\mathcal{L}(v) = \mathtt{R}(t)$ with a subgraph $v_1 \to v_2$ where $v_1 = \neg\mathtt{R}(t)$ and $v_2 = \mathtt{R}(t)$. If $\mathcal{S}(v) = T$, then $\lambda(v_1) = L$ and $\lambda(v_2) = W$. Otherwise, $\lambda(v_1) = W$ and $\lambda(v_2) = L$.
- For each rule node v set $\lambda(v) = L$ if $\mathcal{S}(v) = T$ and $\lambda(v) = W$ otherwise. For each goal node v set $\lambda(v) = W$ if $\mathcal{S}(v) = T$ and $\lambda(v) = L$ otherwise.
- Edges between rule and goal nodes are preserved unmodified. Edges from tuple nodes to rule nodes stem from the positive tuple node. Consider a goal node v with $\mathcal{S}(v) = T$ that is connected to a tuple node v'. If $\mathcal{S}(v) = \mathcal{S}(v')$ (positive goals), then v is connected to the negative tuple node derived from v'; if $\mathcal{S}(v) \neq \mathcal{S}(v')$ then v is connected to the positive tuple node.

$\text{Tr}_{\Gamma\to\text{Expl}}$. The reverse translation is defined as below:

- Remove all EDB fact nodes.
- Replace each subgraph $v_1 \to v_2$ where v_1 is labeled $\neg\mathtt{R}(t)$ and v_2 is labeled $\mathtt{R}(t)$ with a tuple node v with $\mathcal{L}(v) = \mathtt{R}(t)$ and $\mathcal{S}(v) = T$ if $\lambda(v_2) = W$ and $\mathcal{S}(v) = F$ otherwise. All incoming and outgoing edges to/from v_1 and v_2 are connected to v.

– For every rule node v labeled $r(\vec{c})$, set $\mathcal{S}(v) = T$ if $\lambda(v) = L$ and $\mathcal{S}(v) = F$ otherwise.
– For every goal node v labeled $g_i^j(\vec{c})$, set $\mathcal{S}(v) = T$ if $\lambda(v) = W$ and $\mathcal{S}(v) = F$ otherwise.

Theorem 1. *Let P be a program, I a database instance, and ψ a* PQ*. We have:*

$$\text{TR}_{\Gamma\to\text{EXPL}}(\Gamma(P,\psi,I)) = \text{EXPL}(P,\psi,I)$$
$$\text{TR}_{\text{EXPL}\to\Gamma}(\text{EXPL}(P,\psi,I)) = \Gamma(P,\psi,I)$$

Proof. By construction, the label of a tuple node in a provenance graph is T if the tuple exists and F otherwise. Rule nodes are labeled T if the rule derivation is successful and F otherwise. As shown in [24], a tuple node in a provenance game is labeled W if the tuple exists and L otherwise. A rule node is labeled L if the derivation is successful and W else. Thus, based on the definition of $\text{TR}_{\Gamma\to\text{EXPL}}$ and $\text{TR}_{\text{EXPL}\to\Gamma}$, tuple nodes are translated correctly. An analogous argument holds for rule nodes and goal nodes. What remains to be shown is that nodes are correctly connected in the result of $\text{TR}_{\Gamma\to\text{EXPL}}$ and $\text{TR}_{\text{EXPL}\to\Gamma}$. In both provenance games and our provenance graphs, IDB tuple nodes are connected to the rule nodes deriving them. These edges are preserved by both mappings. Rule nodes representing successful derivations are connected to all corresponding goal nodes and, for unsuccessful derivations, only to goals that are failed. This is true for both provenance games and our provenance graphs. Both mappings preserve these edges. In provenance graphs, goal nodes are connected to the tuple node corresponding to the grounded goal. Finally, in provenance games, there are additional nodes (negated tuple nodes and EDB fact nodes). These nodes always co-occur with a positive tuple nodes in a fixed graph fragment, e.g., $\neg\mathtt{R}(t) \to \mathtt{R}(t) \to r_R(t)$. Mapping $\text{TR}_{\text{EXPL}\to\Gamma}$ creates these fragments. Mapping $\text{TR}_{\Gamma\to\text{EXPL}}$ collapses these fragments and reroutes the incoming and outgoing edges of such a subgraph to the tuple node the graph fragment is mapped to. Since explanations and game explanations are defined based on connectivity, this concludes the proof. □

Example 4. *Consider rule r_2 in Fig. 4a computing which cities can be reached from another city through a path of length 3. The provenance game and provenance graph for $\mathtt{Q_{3hop}}(s,s)$ are shown in Fig. 4d and Fig. 4e, respectively. In the provenance graph, goal nodes are directly connected to tuple nodes. In the game, they are represented as positive and negative tuple nodes and EDB fact nodes (the lower three levels). That is, every subgraph $\neg T(X,Y) \to T(X,Y) \to r_T(X,Y)$ in Fig. 4d is equivalently encoded as a single tuple node $T(X,Y)$ in Fig. 4e. Both graphs record the 3 paths of length 3 which start and end in Seattle: 1) $s \to s \to s \to s$, 2) $s \to c \to s \to s$, and 3) $s \to s \to c \to s$.*

6 Semiring Provenance for Positive Queries

The semiring annotation model [22,15,18] is widely accepted as a provenance model for positive queries. An interesting question is how our model compares to provenance polynomials (semiring $\mathbb{N}[X]$), the most general form of annotation in the semiring model. It was shown in [24] that, for positive queries, the result of a query annotated with semiring $\mathbb{N}[X]$ can be extracted from the provenance game by applying a graph transformation. The equivalence shown in Sec. 5 extends this result to our provenance graph model. That being said, we develop simplified versions of our graph model to directly support less informative provenance semirings such as Lineage which only tracks data-dependencies between input and output tuples. We now introduce the semiring annotation framework for positive queries and its use in provenance tracking and, then, explain our simplified provenance graph types.

6.1 $\mathcal{K}$-relations

In the semiring framework, relations are annotated with elements from a commutative semiring. A commutative semiring is a structure $\mathcal{K} = (K, +_\mathcal{K}, \cdot_\mathcal{K}, 0_\mathcal{K}, 1_\mathcal{K})$ over a set K where the addition and multiplication operations are associative, commutative, and have a neutral element ($0_\mathcal{K}$ and $1_\mathcal{K}$, respectively). Furthermore, multiplication with zero yields zero and multiplication distributes over addition. A relation annotated with the elements of a semiring $\mathcal{K}$ is called a $\mathcal{K}$-relation. Operators of positive relational algebra ($\mathcal{RA}^+$) for $\mathcal{K}$-relations compute annotations for tuples in their output by combining annotations from their input using the operations of the semiring. Intuitively, multiplication represents conjunctive use of inputs (as in a join) whereas addition represents alternative use of inputs (as in a union or projection). We are interested in $\mathcal{K}$-relations, because it was shown that many provenance types can be expressed as semiring annotations.

Semiring homomorphisms are important for our purpose since they allow us to translate between different provenance semirings and understand their relative informativeness. A semiring homomorphism $h : \mathcal{K}_1 \to \mathcal{K}_2$ is a function from K_1 to K_2 that respects the operations of semirings, e.g., $h(k_1 +_{\mathcal{K}_1} k_2) = h(k_1) +_{\mathcal{K}_2} h(k_2)$. As shown in [14], if there exists a surjective homomorphism

between one provenance semiring $\mathcal{K}_1$ and another semiring $\mathcal{K}_2$, then $\mathcal{K}_1$ is more informative than $\mathcal{K}_2$ (see [18] for the technical details justifying this argument). We introduce several provenance semirings below and explain the homomorphisms that link the most informative semiring ($\mathbb{N}[X]$) to less informative semirings.

$(\mathbb{N}[X], +, \cdot, 0, 1)$: The elements of semiring $\mathbb{N}[X]$ are polynomials with natural number coefficients and exponents over a set of variables X representing tuples. Any polynomial can be written as a sum of products by applying the equational laws of semirings, e.g., the provenance polynomial for query result $\mathtt{Q}_{\mathtt{3hop}}(s,s)$ is p^3+2pqr (Fig. 4a). An important property of $\mathbb{N}[X]$ is that there exist homomorphisms from $\mathbb{N}[X]$ to any other semiring.

$(\mathsf{PosBool}(X), +, \cdot, 0, 1)$: The elements of $\mathsf{PosBool}(X)$ are derived from $\mathbb{N}[X]$ by making both addition and multiplication idempotent and applying an additional equational law: $x + x \cdot y = x$. An element from $\mathsf{PosBool}(X)$ can be encoded as a set of sets of variables with the restriction that every inner set k is minimal, i.e., there is no other inner set k' that is a subset of k. For example, the provenance polynomial $p^3 + 2pqr$ of $\mathtt{Q}_{\mathtt{3hop}}(s,s)$ is simplified as follows: $p^3 + 2pqr = p + pqr = p$.

$(\mathsf{Which}(X), +, \cdot, 0, 1)$: In the $\mathsf{Which}(X)$ semiring, addition is equivalent to multiplication: $x + y = x \cdot y$ for $x, y \notin \{0, 1\}$, and both addition and multiplication are idempotent. This semiring has sometimes also be called the Lineage semiring. Alternatively, the semiring can be defined over the powerset of the set of variables X [5].

Other semirings of interest are $(\mathbb{B}[X], +, \cdot, 0, 1)$ which is derived from $\mathbb{N}[X]$ by making addition idempotent ($x+x \equiv x$), semiring $(\mathsf{Trio}(X), +, \cdot, 0, 1)$ where multiplication is idempotent ($x \cdot x \equiv x$), and $(\mathsf{Why}(X), +, \cdot, 0, 1)$ where both addition and multiplication are idempotent.

6.2 $\mathcal{K}$-explanations

We now introduce simplified versions of our provenance graphs that each corresponds to a certain provenance semiring. Given a positive query P, PQ ψ, and database I, we use $\textsc{Expl}_{\mathcal{K}}(P, \psi, I)$ to denote a $\mathcal{K}$-explanation for ψ. A $\mathcal{K}$-explanation is a provenance graph that encodes the $\mathcal{K}$-provenance of all query results from $\textsc{Match}(\psi)$, i.e., the set of answers the user is interested in. In the following, we first show how to extract $\mathbb{N}[X]$ from our provenance graph. Then, for each homomorphism implementing the derivation of a less informative provenance model from a more informative provenance model in the semiring framework, there is a corresponding graph transformation over our provenance graphs that maps $\textsc{Expl}_{\mathbb{N}[X]}(P, \psi, I)$ to $\textsc{Expl}_{\mathcal{K}}(P, \psi, I)$. The following theorem shows that we can reuse the existing mapping from provenance games to provenance polynomials by composing it with the mapping $\textsc{Tr}_{\textsc{Expl}\to\Gamma}$.

Theorem 2. *Let $\textsc{Tr}_{\textsc{Expl}\to\mathbb{N}[X]}$ denote the function $\textsc{Tr}_{\Gamma\to\mathbb{N}[X]} \circ \textsc{Tr}_{\textsc{Expl}\to\Gamma}$. Given a positive input program P, database instance I, and tuple $t \in P(I)$, denote by $\mathbb{N}[X](P, I, t)$ the $\mathbb{N}[X]$ annotation of t over an abstractly tagged version of I (each tuple t is annotated with a unique variable x_t). Then,*

$$\textsc{Tr}_{\textsc{Expl}\to\mathbb{N}[X]}(\textsc{Expl}(P, I, t)) = \mathbb{N}[X](P, I, t)$$

Proof. It has already proven in [24] that provenance games generalize $\mathbb{N}[X]$ (provenance polynomial can be extracted from games). That is, using our notation,

$$\textsc{Tr}_{\Gamma\to\mathbb{N}[X]}(\Gamma(P, I, t)) = \mathbb{N}[X](P, I, t)$$

To prove Theorem 2, we have to show that the game explanation $\Gamma(P, I, t)$ is equivalent to our explanation $\textsc{Expl}(P, I, t)$. In Theorem 1, we have proven that our provenance graphs are equivalent to provenance games (there exists a lossless transformation). By applying the transformation $\textsc{Tr}_{\textsc{Expl}\to\Gamma}$ to the formula above, we get

$$\textsc{Tr}_{\Gamma\to\mathbb{N}[X]}(\textsc{Tr}_{\textsc{Expl}\to\Gamma}(\textsc{Expl}(P, I, t))) = \mathbb{N}[X](P, I, t)$$

Thus, it is immediate that the function $\textsc{Tr}_{\Gamma\to\mathbb{N}[X]} \circ \textsc{Tr}_{\textsc{Expl}\to\Gamma}$ is correct and, thus, the claim holds. □

Consider the explanation for $\textsc{Why}\,\mathtt{Q}_{\mathtt{3hop}}(s,s)$ shown in Fig. 4e. Recall that there are three options for reaching Seattle from Seattle with two intermediate stops corresponding to three derivations of $\mathtt{Q}_{\mathtt{3hop}}$ using rule r_2. These three derivations are shown in the provenance graph, e.g., $r_2(s,s,s,s)$ is the derivation that uses the local train connection inside Seattle three times. Annotating the train connections with variables p, q, and r as shown in Fig. 4a and ignoring rule information encoded in the graph, the provenance encoded by our model is a bag (denoted as [ ]) of lists (denoted as ( )) of these variables. Each list corresponds to a rule derivation where variables are ordered according to the order of their occurrence in the body of the rule. For instance, (q, r, p) corresponds to taking a train from Seattle to Chicago (q), then from Chicago to Seattle (r), and finally a local connection inside Seattle (p). We now illustrate the graph transformations yielding $\mathcal{K}$-explanations from $\textsc{Expl}_{\mathbb{N}[X]}$ based on this example.

Semiring $\mathbb{N}[X]$. In Fig. 4e, we (1) replace rule nodes with multiplication (i.e., $r_2(s,s,s,s) \to \cdot$) and (2) replace goal nodes with addition (e.g., $g_4^1(s,s) \to +$) to generate a graph that encodes $\mathbb{N}[X]$ as shown in Fig. 4b (denoted as $\textsc{Expl}_{\mathbb{N}[X]}$). Applying this transformation, the rule instantiation $r_2(s,s,c,s)$ deriving result tuple $\mathtt{Q}_{\mathtt{3hop}}(s,s)$ can no longer be distinguished from

$r_2(s,s,s,c)$, because they are connected to the same tuple nodes. The only information retained is which arguments are used how often by a rule (labelled with $\cdot$). To extract $\mathbb{N}[X]$, we (1) replace label leaf nodes with their annotation from Fig. 4a (e.g., $T(s,s)$ is replaced with p) and (2) replace IDB tuple nodes with addition.

Semiring $\mathsf{PosBool}(X)$**.** $\text{Expl}_{\mathsf{PosBool}(X)}$ (Fig. 4f) is computed from $\text{Expl}_{\mathbb{N}[X]}$ by first collapsing rule nodes if the subgraphs rooted at these rule nodes are isomorphic and dropping all the goal nodes. Then, "$\cdot$" nodes are removed if one or more subgraphs rooted at children of such a node is isomorphic to the subgraphs rooted at the children of another "$\cdot$" node. Applying this process to our example, after the first step, two "$\cdot$" nodes (one connects $\mathtt{Q}_{\mathtt{3hop}}(s,s)$ to p and the other for each p, q, and r) exist in the graph corresponding to p and $p \cdot q \cdot r$. In the second step, $p \cdot q \cdot r$ is removed because it contains p ($\mathtt{T}(s,s)$) as a subgraph.

Semiring $\mathsf{Which}(X)$**.** The semiring $\mathsf{Which}(X)$ (aka Lineage) is reached by collapsing all intermediate nodes and directly connecting tuple nodes (e.g., $\mathtt{Q}_{\mathtt{3hop}}(s,s)$) with other tuple nodes (e.g., $\mathtt{T}(s,s)$) as shown in Fig. 4c.

$\text{Expl}_{\mathbb{B}[X]}$ and $\text{Expl}_{\mathsf{Trio}(X)}$ are derived from $\text{Expl}_{\mathbb{N}[X]}$ by collapsing isomorphic subgraphs rooted at rule nodes and by dropping all the goal nodes, respectively. The combination of these transformations achieves $\text{Expl}_{\mathsf{Why}(X)}$.

7 Semiring Provenance for FO Model Checking

The semiring framework was recently extended for capturing provenance of first-order (FO) model checking [37, 13]. We now study the relationship of our model to semiring provenance for FO queries. Based on the observation first stated in [24] (provenance for FO queries and, thus, also FO logic, naturally supports missing answers), the authors explain missing answers based on FO provenance [41]. Another interesting aspect of [13] is that it allows some facts to be left undetermined (their truth is undecided). This enables how-to queries [29], i.e., given an expected outcome, which possible world compatible with the undecided facts would produce this outcome. In this section, we first introduce the model from [13], then demonstrate how our approach can be extended to support undetermined truth values. Finally, we show how the annotation computed by the approach presented in [13] for a FO formula φ can be efficiently extracted from the provenance graph generated by our approach for a query Q_φ which is derived from φ through a translation $\text{Tl}_{\varphi \to Q}$.

7.1 K-Interpretations and Dual Polynomials

In [37,18], the authors define semiring provenance for formulas in FO logic. Let A be a domain of values. We use ν to denote an assignment of the free variables of φ to values from A. Given a so-called $\mathcal{K}$-interpretation π which is a function mapping positive and negative literals to annotations from $\mathcal{K}$, the annotation of a formula $\pi[\![\varphi]\!]_\nu$ for a given valuation ν is derived using the rules below. For sentences, i.e., formulas without free variables, we omit the valuation and write $\pi(\varphi)$ to denote $\pi[\![\varphi]\!]_\nu$ for the empty valuation ν. Furthermore, **op** is used to denote a comparison operator (either $=$ or $\neq$).

$$\pi[\![R(\mathbf{x})]\!]_\nu = \pi(R(\nu(\mathbf{x}))) \quad \pi[\![\neg\, R(\mathbf{x})]\!]_\nu = \pi(\neg\, R(\nu(\mathbf{x})))$$

$$\pi[\![x \,\mathbf{op}\, y]\!]_\nu = \text{ if } \nu(x) \,\mathbf{op}\, \nu(y) \text{ then 1 else 0} \quad \pi[\![\neg\varphi]\!]_\nu = \pi[\![\mathbf{nnf}(\varphi)]\!]_\nu$$

$$\pi[\![\varphi_1 \vee \varphi_2]\!]_\nu = \pi[\![\varphi_1]\!]_\nu + \pi[\![\varphi_2]\!]_\nu \quad \pi[\![\varphi_1 \wedge \varphi_2]\!]_\nu = \pi[\![\varphi_1]\!]_\nu \cdot \pi[\![\varphi_2]\!]_\nu$$

$$\pi[\![\exists x\, \varphi]\!]_\nu = \sum_{a \in A} \pi[\![\varphi]\!]_{\nu[x \mapsto a]} \quad \pi[\![\forall x\, \varphi]\!]_\nu = \prod_{a \in A} \pi[\![\varphi]\!]_{\nu[x \mapsto a]}$$

Both conjunction and universal quantification correspond to multiplication, and annotations of positive and negative literals are read from π. This model deals with negation as follows. A negated formula is first translated into negation normal form (nnf). A formula in **nnf** does not contain negation except for negated literals. Any formula can be translated into this form using DeMorgan rules, e.g., $\neg\,(\forall x\, \varphi) \equiv (\exists x \neg\, \varphi)$. By pushing negation to the literal level using **nnf**, and annotating both positive and negative literals, the approach avoids extending the semiring structure with an explicit negation operation.

Provenance tracking for FO formula has to take into account the dual nature of the literals. The solution presented in [13,37] is to use polynomials over two sets of variables: variables from X and $\bar{X}$ are exclusively used to annotate positive and negative literals, respectively. For any variable $x \in X$, there exists a corresponding variable $\bar{x} \in \bar{X}$ and vice versa. Furthermore, if x annotates $R(\mathbf{a})$, then $\bar{x}$ can only annotate $\neg\, R(\mathbf{a})$ (and vice versa). The semiring of dual indeterminate polynomials is then defined as the structure generated by applying the congruence $x \cdot \bar{x} = 0$ to the polynomials from $\mathbb{N}[X \cup \bar{X}]$. The resulting structure is denoted by $\mathbb{N}[X, \bar{X}]$. Intuitively, this congruence encodes the standard logic equivalence $R(\mathbf{a}) \wedge \neg\, R(\mathbf{a}) \equiv false$. Importantly, in a $\mathbb{N}[X, \bar{X}]$-interpretation π, we can decide which facts are true/false and whether to track provenance for these facts. Furthermore, we can leave the truth of some literals undetermined. Below, we show all feasible combinations for annotating $R(\mathbf{a})$ and $\neg\, R(\mathbf{a})$ in π and their meaning. For instance, if we annotate $R(\mathbf{a})$ with 1 or 0, this corresponds to asserting the fact $R(\mathbf{a})$, but not tracking provenance for it. By setting

$R(\mathbf{a}) = x$ and $\neg R(\mathbf{a}) = \bar{x}$, we leave the truth of $R(\mathbf{a})$ undecided. Note that $R(\mathbf{a}) = 0$ and $\neg R(\mathbf{a}) = 0$ (as well as $R(\mathbf{a}) = 1$ and $\neg R(\mathbf{a}) = 1$) are not considered here since they lead to incompleteness (inconsistency).

$\pi(R(\mathbf{a})) = 1 \quad \pi(\neg R(\mathbf{a})) = 0$ (true, no provenance)
$\pi(R(\mathbf{a})) = 0 \quad \pi(\neg R(\mathbf{a})) = 1$ (false, no provenance)
$\pi(R(\mathbf{a})) = x \quad \pi(\neg R(\mathbf{a})) = 0$ (true, track provenance)
$\pi(R(\mathbf{a})) = 0 \quad \pi(\neg R(\mathbf{a})) = \bar{x}$ (false, track provenance)
$\pi(R(\mathbf{a})) = x \quad \pi(\neg R(\mathbf{a})) = \bar{x}$ (undetermined)

Consider a sentence φ.[3] The annotation $\pi(\varphi)$ computed for φ over π with undetermined facts represents a set of possible models for φ. By choosing for each undetermined fact $R(\mathbf{a})$ in $\pi(\varphi)$ whether it is true or not, we "instantiate" one possible model for φ. By encoding a set of possible models, $\pi(\varphi)$ allows for reverse reasoning: we can find models that fulfill certain properties from the set of models encoded by $\pi(\varphi)$.

Example 5. *Reconsider query r_1 from Fig. 1. Assume that we want to determine what effect building a direct train connection from New York to Seattle would have on the query result $\mathtt{Q}(n, s)$. Thus, we make the assumption that the database instance is as in Fig. 1 with the exception that we keep $\mathtt{T}(n, s)$ undetermined. In first-order logic, r_1 is expressed as: $\mathtt{only2hop}(x, y) \equiv \exists z(\mathtt{T}(x, z) \wedge \mathtt{T}(z, y) \wedge \neg\mathtt{T}(x, y))$ and fact $\mathtt{Q}(n, s)$ as: $\varphi \equiv \mathtt{only2hop}(n, s)$. The database when keeping $\mathtt{T}(n, s)$ undetermined is encoded as a $\mathbb{N}[X, \bar{X}]$-interpretations π which assigns variables to positive literals as shown in Fig. 1 (the corresponding negated literals are annotated with 0). $\pi(T(n, s)) = v$, $\pi(\neg T(n, s)) = \bar{v}$, and we annotate all remaining positive literals with 0 and negative literals with 1. Computing $\pi(\varphi)$ using the rules above, we get $(t \cdot s \cdot \bar{v}) + (u \cdot r \cdot \bar{v})$. There are two ways of deriving the query result $\mathtt{Q}(n, s)$ which both depend on the absence of a direct train connection from New York to Seattle ($\bar{v}$). Now if we decide to introduce such a connection, we can evaluate the effect of this choice by setting $\bar{v} = 0$ in the provenance polynomial above (the absence of this connection has been refuted), i.e., we get $(t \cdot s \cdot 0) + (u \cdot r \cdot 0) = 0$. Thus, if we were to introduce such a connection, then $\mathtt{Q}(n, s)$ would no longer be a result.*

7.2 Supporting Undeterminism in Provenance Graphs

Supporting undetermined facts in our provenance model is surprisingly straight-forward. We introduce a new label U which is used to label nodes whose success/failure (existence/absence) is undetermined. To account for this new label, we amend the rules for determining connectivity and node labeling as follows:

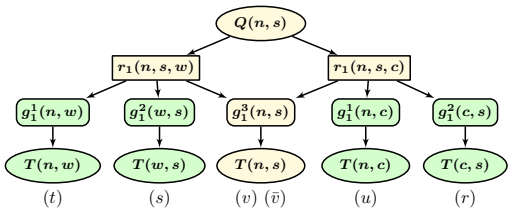


Fig. 5: Provenance graph for WHY $\mathtt{Q}(n, s)$ when $T(n, s)$ is left undetermined

- For a goal node v_g (no matter whether positive or negative) that is connected to a tuple node v_t with $\mathcal{S}(v_t) = U$, we set $\mathcal{S}(v_g) = U$ (goals corresponding to undetermined tuples are undetermined).
- A rule node is succesful (T) if all its goals are successful, a rule node is failed if some of its goals are failed (F), and finally a rule node is undetermined (U) if at least one of its goals is undetermined and none of its goals are failed. Successful rule nodes are connected to all goals, failed rule nodes to failed and undetermined goals (these may provide further justification for the failure), and undetermined rule nodes to all goals (successful and undetermined goals will determine success of the rule nodes under choices).
- An IDB tuple exists (T) if at least one of its rule derivation is successful. It is connected to all successful and undetermined rule derivations (these may provide additional justifications under certain choices for undetermined facts). An IDB tuple is absent (F) if all of its rule derivations fail. Finally, an IDB tuple is undetermined (U) if at least one of its rule derivations is undetermined and none is successful. Undetermined tuple nodes are connected to all their rule derivations (failed ones may be additional justifications for absence while undetermined ones may justify either existence or absence).

Example 6. *Consider Example 5 in our extended provenance graph model. Let $v_{n,s}$ be the node corresponding to $\mathtt{T}(n, s)$. If we set $\mathcal{S}(v_{n,s}) = U$ to indicate that $\mathtt{T}(n, s)$ should be considered as undetermined, then we get the provenance graph in Fig. 5. Our approach correctly determines that under this assumption the truth of $\mathtt{Q}(n, s)$ is undetermined and that there are two potential derivations of this result which also are undetermined, because they depend on existing tuples as well as the undetermined tuple $\mathtt{T}(n, s)$. To evaluate the effect of choosing $\mathtt{T}(n, s)$ to be true or false, we would set $\mathcal{S}(v_{n,s}) = T$ or $\mathcal{S}(v_{n,s}) = F$ and propagate the effect of this change bottom-up through-out the provenance graph.*

[3] We only restrict the discussion to sentences for simplicity. The arguments here also hold for formulas with free variables.

$$\frac{\varphi := \exists x : \varphi_1}{Q_\varphi(\text{FREE}(\varphi)) :- Dom(x), Q_{\varphi_1}(\text{FREE}(\varphi_1))} \quad (1) \qquad \frac{\varphi := \neg R(\mathbf{x}), \text{FREE}(\varphi) = \{x_1, \ldots, x_n\}}{Q_\varphi(\text{FREE}(\varphi)) :- Dom(x_1), \ldots, Dom(x_n), \neg R(\mathbf{x})} \quad (2) \qquad \frac{\varphi := R(\mathbf{x})}{Q_\varphi(\text{FREE}(\varphi)) :- R(\mathbf{x})} \quad (3)$$

$$\frac{\varphi := \varphi_1 \vee \varphi_2, \text{FREE}(\varphi_1) = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}, \text{FREE}(\varphi_2) = \{x_1, \ldots, x_n, z_1, \ldots, z_l\}}{\begin{array}{l} Q_\varphi(\text{FREE}(\varphi)) :- Dom(z_1), \ldots, Dom(z_k), Q_{\varphi_1}(\text{FREE}(\varphi_1)) \\ Q_\varphi(\text{FREE}(\varphi)) :- Dom(y_1), \ldots, Dom(y_m), Q_{\varphi_2}(\text{FREE}(\varphi_2)) \end{array}} \quad (4) \qquad \frac{\varphi := x \,\mathbf{op}\, y}{Q_\varphi(x, y) :- Dom(x), Dom(y), x \,\mathbf{op}\, y} \quad (5)$$

$$\frac{\varphi := \forall x : \varphi_1, \text{FREE}(\varphi) = \{x_1, \ldots, x_n\}}{\begin{array}{l} Q_\varphi(\text{FREE}(\varphi)) :- Dom(x_1), \ldots, Dom(x_k), \neg Q_{\varphi'}(\text{FREE}(\varphi)) \\ Q_{\varphi'}(\text{FREE}(\varphi)) :- Dom(x), Dom(x_1), \ldots, Dom(x_n), \neg Q_{\varphi_1}(\text{FREE}(\varphi_1)) \end{array}} \quad (6) \qquad \frac{\varphi := \varphi_1 \wedge \varphi_2}{Q_\varphi(\text{FREE}(\varphi)) :- Q_{\varphi_1}(\text{FREE}(\varphi_1)), Q_{\varphi_2}(\text{FREE}(\varphi_2))} \quad (7)$$

Fig. 6: Translating a first-order formula φ into a first-order query Q_φ

Importantly, the provenance graph captured for an instance with undetermined facts is sufficient for evaluating the effect of setting any of these undetermined facts to false or true. That is, just like it is not necessary to reevaluate the semiring annotation of a formula to evaluate the impact of such a choice, it is also not necessary to recapture provenance in our model to evaluate a choice. For lack of space, we are not discussing the details of a corresponding extension for provenance games, but still would like to remark that undetermined facts correspond to draws in the game (neither player has a winning strategy). In the type of two-player games employed in provenance games, draws are caused by cycles in the game graph. To leave the existence of an EDB tuple undetermined, we introduce an EDB fact node for the tuple and add a self-edge to this node which causes the tuple node to be a draw in the game.

7.3 From First-order Formulas to Datalog

We now present a translation $\text{TL}_{\varphi \to Q}$ from FO formulas φ to boolean Datalog queries Q_φ. The query generated based on a formula φ is equivalent to the formula in the following sense: if φ evaluates to true for a model, then $Q_\varphi(I)$ returns true. Here, I is the instance that contains precisely the tuples corresponding to literals that are true in the model. We assume that the free variables of a formula (the variables not bound by any quantifier) are distinct from variable names bound by quantifiers and that no two quantifiers bind a variable of the same name. This can be achieved by renaming variables in a formula that does not fulfill this condition. For example, $(\forall x\, R(x, y)) \wedge (\exists y\, S(y))$ does not fulfill this condition, but the equivalent formula $(\forall x\, R(x, y)) \wedge (\exists z\, S(z))$ does. We also assume an arbitrary, but fixed, total order $<_{Var}$ over variables that appear in formulas. We use $\text{FREE}(\varphi)$ to denote the list of free variables of a formula φ ordered increasingly by $<_{Var}$. For instance, for $\varphi := \forall x : R(x, y)$ we have $\text{FREE}(\varphi) = \{y\}$. Our translation $\text{TL}_{\varphi \to Q}$ takes as input a formula φ and outputs a Datalog program with an answer predicate Q_φ. The translation rules are shown in Fig. 6. Each rule translates one construct (e.g., a quantifier) and outputs one or more Datalog rules. The Datalog program generated by the translation for an input φ is the set of Datalog rules generated by applying the rules from Fig. 6 to all sub-formulas of φ. Here, we assume the existence of a unary predicate *Dom* whose extension is the domain A. Most translation rules are straight-forward and standard. Logical operators are translated into their obvious counterpart in Datalog, e.g., a conjunction $\varphi_1 \wedge \varphi_2$ is translated into a rule with two body atoms Q_{φ_1} and Q_{φ_2}. The rules generated for a formula φ return the formula's free variables to make them available to formulas that use φ. For instance, since Datalog does not support universal quantification directly, we have to simulate it using double negation ($\forall x\, \varphi$ is rewritten as $\neg \exists x \neg \varphi$). Disjunctions are turned into unions. The complexity of the rule for disjunction stems from the fact that, in $\varphi_1 \vee \varphi_2$, the sets of free variables for φ_1 and φ_2 may not be the same. To make them union compatible, use $\text{FREE}(\varphi)$ as the arguments of the heads of the rules for both φ_1 and φ_2, and add additional goals *Dom* to ensure that these rules are safe.

Example 7. *Consider a directed graph encoded as its edge relation* R. *The formula* $\varphi := \forall x \exists y\, \text{R}(x, y)$ *checks whether all nodes in the graph have outgoing edges. Let* $\varphi_1 = \exists y\, \text{R}(x, y)$ *and* $\varphi_2 = \text{R}(x, y)$. *Translating this formula, we get:*

$$Q_\varphi() :- \neg Q_{\varphi'}() \qquad Q_{\varphi'}() :- Dom(x), \neg Q_{\varphi_1}(x)$$

$$Q_{\varphi_1}(x) :- Dom(y), Q_{\varphi_2}(x, y) \qquad Q_{\varphi_2}(x, y) :- \text{R}(x, y)$$

7.4 From Graphs to FO Semiring Provenance

Given a formula φ in negation normal form (nnf) and a $\mathbb{N}[X, \bar{X}]$-interpretation π, we now demonstrate how to extract $\pi[\![\varphi]\!]_\nu$ from the subgraph of the provenance graph generated based on π over $\text{TL}_{\varphi \to Q}(\varphi)$ rooted at the tuple node $Q_\varphi(\nu(\text{FREE}(\varphi)))$. First, we apply $\text{TL}_{\varphi \to Q}(\varphi)$ to compute Q_φ. Then, we generate an instance I_π where the existence of a tuple corresponding to a literal $\text{R}(\mathbf{a})$ is determined based on the truth value of this literal encoded by its annotation $\pi(\text{R}(\mathbf{a}))$. A tuple $\text{R}(\mathbf{a})$ exists

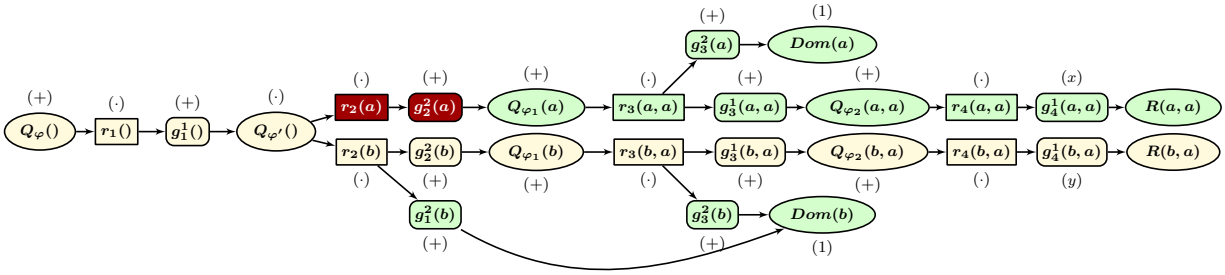


Fig. 7: Provenance graph for query Q_φ based on $\varphi := \forall x\, \exists y\, \mathtt{R}(x,y)$ when $R(a,a)$ is true ($\pi(\mathtt{R}(a,a)) = x$ and $\pi(\neg\mathtt{R}(a,a)) = 0$) and $R(b,a)$ is left undetermined ($\pi(\mathtt{R}(b,a)) = y$ and $\pi(\neg\mathtt{R}(b,a)) = \bar{y}$). Note that $\varphi_1 := \exists y\, R(x,y)$ and $\varphi_2 := R(x,y)$. The result of $\mathrm{TR}_{\mathrm{EXPL}\to\mathbb{N}[X,\bar{X}]}$ shown besides the nodes encodes the dual polynominal $\pi(\varphi) = x \cdot y$.

in I_π if $\pi(\mathtt{R}(\mathbf{a})) = x$ or $\pi(\mathtt{R}(\mathbf{a})) = 1$ and $\pi(\neg\mathtt{R}(\mathbf{a})) = 0$, the tuple is missing if $\pi(\neg\mathtt{R}(\mathbf{a})) = \bar{x}$ or $\pi(\neg\mathtt{R}(\mathbf{a})) = 1$ and $\pi(\mathtt{R}(\mathbf{a})) = 0$, and the tuple's existence is undetermined if $\pi(\mathtt{R}(\mathbf{a})) = x$ and $\pi(\neg\mathtt{R}(\mathbf{a})) = \bar{x}$. Note that this corresponds to the truth value according to the 5 cases we have discerned in Sec. 7.1.

Next, we generate the provenance graph $\mathcal{PG}(Q_\varphi, I_\pi)$. If the formula has free variables, then the provenance graph will contain multiple tuple nodes $Q_\varphi(\nu(\mathrm{FREE}(\varphi)))$, one for each valuation ν of the free variables, and the subgraph rooted at one such tuple node encodes $\pi[\![\varphi]\!]_\nu$. By applying a function $\mathrm{TR}_{\mathrm{EXPL}\to\mathbb{N}[X,\bar{X}]}$ (defined in the following), we translate the subgraph rooted at tuple $Q_\varphi(\nu(\mathrm{FREE}(\varphi)))$ in $\mathcal{PG}(Q_\varphi, I_\pi)$ into $\pi[\![\varphi]\!]_\nu$.

The function $\mathrm{TR}_{\mathrm{EXPL}\to\mathbb{N}[X,\bar{X}]}$ replaces nodes in the provenance graph with nodes labeled as "+", "·", and annotations of literals. The polynomial $\pi[\![\varphi]\!]_\nu$ can then be read from the graph generated by $\mathrm{TR}_{\mathrm{EXPL}\to\mathbb{N}[X,\bar{X}]}$ through a top-down traversal. Intuitively, the translation can be explained as follows. Datalog rules are a conjunction of atoms and, thus, are replaced with multiplication. There may exist multiple ways that derive an IDB tuple through the rules of query. That is, IDB tuple nodes represent addition. The exception is IDB tuples that are used in a negated fashion which are replaced with multiplication, because, for the goal to succeed, all derivations of the tuple have to fail. Note that a tuple is used in a negated way if there is an odd number of negated goals on the path between the root of the provenance graph and IDB tuple node. In a program produced by our translation rules, this can only be the case for tuples that correspond to head predicate of a rule computing the $\neg\exists$ part of the translation of a universal quantification.

$\mathrm{TR}_{\mathrm{EXPL}\to\mathbb{N}[X,\bar{X}]}$ consists of the following steps:

1. Replace tuple nodes $Dom(\mathbf{x})$ with 1.
2. A goal node connected to an EDB tuple node representing a literal $\mathtt{R}(\mathbf{a})$ is replaced by $\pi(\mathtt{R}(\mathbf{a}))$ if the goal is positive and $\pi(\neg\mathtt{R}(\mathbf{a}))$ otherwise.
3. Next, all EDB tuple nodes are removed leaving the goal nodes formerly connected to EDB tuple nodes to be the new leaves of the graph.
4. Rule nodes are replaced with multiplication (·).
5. Next all remaining goal nodes are replaced with addition (+).
6. Finally, nodes v_t corresponding to IDB tuples are replaced with addition with the exception of IDB tuples corresponding to the head predicate ($Q_{\varphi'}$) of the second rule of a translated universal quantification which are replaced with multiplication (·).

Example 8. *Consider the formula and query from Example 7. Assume that $A = \{a, b\}$ and consider that interpretation π which tracks provenance for edge (a,a), keeps $\mathtt{R}(b,a)$ undetermined, and sets all other positive literals to false without provenance tracking, i.e., $\pi(\mathtt{R}(a,a)) = x$, $\pi(\neg\mathtt{R}(a,a)) = 0$, $\pi(\mathtt{R}(b,a)) = y$, $\pi(\neg\mathtt{R}(b,a)) = \bar{y}$, and for all other $\mathtt{R}(\mathbf{a})$ we have $\pi(\mathtt{R}(\mathbf{a})) = 0$ and $\pi(\neg\mathtt{R}(\mathbf{a})) = 1$. That is, in I_π, tuple $\mathtt{R}(a,a)$ exists, tuple $\mathtt{R}(b,a)$'s existence is undetermined, and all other tuples are missing. Fig. 7 shows the provenance graph $\mathcal{PG}(Q_\varphi, I_\pi)$. The truth of the universal quantification in φ is undetermined, because while there exists no $a \in A$ such that $\neg\varphi_1$ for $\nu := (x = a)$ is true (there is an outgoing edge starting at a), the truth of $\neg\varphi_1$ is undetermined for $\nu := (x = b)$ (the existence of edge $\mathtt{R}(b,a)$ is undetermined). The truth of $\exists y\, R(a,y)$ and $\exists y\, \mathtt{R}(b,y)$ is justified by the existing tuples $\mathtt{R}(a,a)$ and $\mathtt{R}(b,a)$, respectively. Applying the translation $\mathrm{TR}_{\mathrm{EXPL}\to\mathbb{N}[X,\bar{X}]}$, we get graph with node labels shown in Fig. 7 which corresponds to the polynomial $1 \cdot x \cdot 1 \cdot 1 \cdot y = x \cdot y = \pi(\varphi)$.*

We are now ready to state the main result of this section: our provenance graphs extended for undetermined facts can encode semiring provenance for first-order (FO) model checking. For simplicity, we only consider sentences, i.e., formulas φ without free variables, but the result also holds for formulas with free variables by only translating a subgraph of the provenance rooted at the IDB tuple node $Q_\varphi(\nu(\mathrm{FREE}(\varphi)))$ which corresponds to the formula $\nu(\varphi)$.

Theorem 3. *Let φ be a formula, π a $\mathbb{N}[X,\bar{X}]$-interpretation, $Q := \text{TL}_{\varphi\to Q}(\varphi)$, and I_π the instance corresponding to π as defined above. Then*

$$\text{TR}_{\text{EXPL}\to\mathbb{N}[X,\bar{X}]}(\mathcal{PG}(Q, I_\pi)) = \pi[\![\varphi]\!]_\nu$$

Proof. We prove the theorem by induction over the structure of the input formula φ for a given valuation ν of $\text{FREE}(\varphi)$. Recall that in I_π the existence of a tuple is determined by the annotations assigned by π to the positive and negated literals corresponding to the tuple. In the following, we will often consider the three cases separately (tuple exists, tuples is missing, tuple's existence is undetermined) separately, because our the graph produced by our provenance model may differ based on which case applies. Furthermore, it will be beneficial to also prove that tuple $Q_\varphi(\nu(\text{FREE}(\varphi)))$ exists/is absent/undetermined iff $\nu(\varphi)$ is true/false/undetermined, because it allows us to assume that the truth value of a sub-formula corresponds to $\nu(\varphi)$. In the following we will use $t_{\varphi,\nu}$ to denote $Q_\varphi(\nu(\text{FREE}(\varphi)))$, i.e., the IDB tuple corresponding to applying the valuation ν to formula φ. Furthermore, for IDB tuple $t_{\varphi,\nu}$ use $g_{\varphi,\nu}$ to denote the subgraph of the provenance graph rooted at $t_{\varphi,\nu}$ and $p_{\varphi,\nu}$ to denote $\text{TR}_{\text{EXPL}\to\mathbb{N}[X,\bar{X}]}(g_{\varphi,\nu})$.

Base case: $\varphi := \mathtt{R}(\mathbf{x})$: Such formulas are translated into programs with a single rule. The head predicate Q_φ for this rule has as arguments the free variables of $\mathbf{x}$ which for a literal $\mathtt{R}(\mathbf{x})$ are all variables of the literal. For any valuation ν, there is one corresponding tuple $Q_\varphi(\nu(\mathbf{x}))$ in the provenance graph which as we have mentioned above we denote as $t_{\varphi,\nu}$. First consider the case that the tuple $t_{\varphi,\nu}$ exists which is the case when literal $\mathtt{R}(\nu(\mathbf{x}))$ is true. Applying $\text{TR}_{\text{EXPL}\to\mathbb{N}[X,\bar{X}]}$ to the subgraph corresponding to such a tuple, the tuple node $Q_\varphi(\nu(\mathbf{x}))$ is mapped to $+$, the rule node to $\cdot$, the goal node connected to the EDB node is replaced with $\pi(\mathtt{R}(\nu(\mathbf{x})))$. The resulting expression tree is $+(\cdot(\pi(\mathtt{R}(\nu(\mathbf{x})))))$ which evaluates to $\pi(\mathtt{R}(\nu(\mathbf{x}))) = \pi[\![\varphi]\!]_\nu$. If the existence of $\mathtt{R}(\nu(\mathbf{x}))$ is undetermined, then the rule deriving the result tuple $t_{\varphi,\nu}$ are also undetermined. However, the resulting expression is still the same. If tuple $\mathtt{R}(\nu(\mathbf{x}))$ is missing we still get a graph with the same structure.

Base case: $\neg\mathtt{R}(\mathbf{x})$: A formula $\neg\mathtt{R}(\mathbf{x})$ is translated into a single rule that "joins" the atom $\neg\mathtt{R}(\mathbf{x})$ with multiple *Dom* atoms (one for each variable in $\mathbf{x}$). The arguments of the head predicate of the rule are all variables in $\mathbf{x}$. For a given valuation ν, there exists one IDB tuple node $t_{\varphi,\nu}$. We consider three cases based on the truth of literal $\mathtt{R}(\nu(\mathbf{x}))$. If the tuple exists then there exists one successful binding of the rule connected through its goals to $|\mathbf{x}|$ copies of *Dom* - one for each value in $\nu(\mathbf{x})$ and to a tuple node $\mathtt{R}(\nu(\mathbf{x}))$. Applying the translation and ignoring redundant additions and multiplications (nodes with a single child) we get an expression $\underbrace{1\cdot\ldots\cdot 1}_{|\mathbf{x}|}\cdot\pi(\neg\mathtt{R}(\nu(\mathbf{x}))) = \pi(\neg\mathtt{R}(\nu(\mathbf{x})) = \pi[\![\varphi]\!]_\nu$. If $\mathtt{R}(\nu(\mathbf{x}))$ is undetermined, then the existence of tuple $t_{\varphi,\nu}$ and the status of the rule deriving it are also undetermined. Thus, the structure of the graph is the same as in the first case and the translation returns the same dual provenance polynomial and we have $\pi(\neg\mathtt{R}(\nu(\mathbf{x})) = \pi[\![\varphi]\!]_\nu$. Finally, if literal $\mathtt{R}(\mathbf{x})$ is false then tuple $t_{\varphi,\nu}$ is absent from the query result and the derivation producing it failed. Since failed derivations are only connected to failed goals, the expression created by $\text{TR}_{\text{EXPL}\to\mathbb{N}[X,\bar{X}]}$ is $\cdot(\pi(\neg\mathtt{R}(\nu(\mathbf{x})) = \pi(\neg\mathtt{R}(\nu(\mathbf{x}))) = \pi[\![\varphi]\!]_\nu$.

Inductive step: We assume that the statement holds for formulas φ_1 and/or φ_2 and show that it also holds for a formula φ.

$\varphi := \varphi_1 \wedge \varphi_2$: We have $\pi[\![\varphi]\!]_\nu = \pi[\![\varphi_1]\!]_\nu \cdot \pi[\![\varphi_2]\!]_\nu$ for valuation ν. The program $\text{TL}_{\varphi\to Q}(\varphi)$ consists of single rule $Q_\varphi(\text{FREE}(\varphi)) :-$ $Q_{\varphi_1}(\text{FREE}(\varphi_1)), Q_{\varphi_1}(\text{FREE}(\varphi_2))$. For a valuation ν, $t_{\varphi,\nu}$ exists if $\varphi_1 \wedge \varphi_2$ evaluates to true. In this case the $t_{\varphi,\nu}$ is connected through a rule and two goal nodes to the subgraphs $g_{\varphi_1,\nu}$ and $g_{\varphi_2,\nu}$. Let $p_i = p_{\varphi_i,\nu}$ for $i \in \{1,2\}$ and $p = p_{\varphi,\nu}$. We have $p = +((+(p_1)) \cdot (+(p_2))) = p_1 \cdot p_2$. By the induction hypothesis $p_i = \pi[\![\varphi_i]\!]_\nu$ and we get $p = \pi[\![\varphi_1]\!]_\nu \cdot \pi[\![\varphi_2]\!]_\nu = \pi[\![\varphi]\!]_\nu$. Node $t_{\varphi,\nu}$ is undetermined, if one input is undetermined and the other is either undetermined or true. In this case we get the same provenance expression. Finally, $t_{\varphi,\nu}$ is missing if at least one of the inputs is missing. In this case the provenance graph only contains a subgraph for missing or undetermined inputs. That is, there are three possible provenance expressions extracted from the graph $p_1 = \pi[\![\varphi_1]\!]_\nu$, $p_2 = \pi[\![\varphi_2]\!]_\nu$, or $p_1 \cdot p_2 = \pi[\![\varphi_1]\!]_\nu \cdot = \pi[\![\varphi_2]\!]_\nu$. We claim that in all three cases $p = 0 = \pi[\![\varphi]\!]_\nu$. Consider a failed input $\pi[\![\varphi_i]\!]_\nu$ for $i \in \{1,2\}$. If φ_i evaluates to false, then $\pi[\![\varphi_i]\!]_\nu = 0$. Then $p_i = 0$, the subgraph g_i is connected to node $t_{\varphi,\nu}$, and it follows that $p = 0$.

$\varphi := \varphi_1 \vee \varphi_2$: We have $\pi[\![\varphi_1 \vee \varphi_2]\!]_\nu = \pi[\![\varphi_1]\!]_\nu + \pi[\![\varphi_2]\!]_\nu$. A disjunction is translated into two rules (a union) corresponding joining Q_{φ_i} for $i \in \{1,2\}$ with a number of copies of *Dom* such that both rules return $\text{FREE}(\varphi)$. Tuple $t_{\varphi,\nu}$ exists as long as one tuple $t_{\varphi_i,\nu}$ for $i \in \{1,2\}$ exists. In this case the tuple will be connected through successful or undetermined derivations to nodes $t_{\varphi_i,\nu}$. Let $p = p_{\varphi,\nu}$ and $p_i = p_{\varphi_i,\nu}$. We have either $p = p_1$, $p = p_2$, or $p = p_1 + p_2$. In the last case trivially we have $p = \pi[\![\varphi]\!]_\nu$. Now consider the case where $p = p_1$ (the case for $p = p_2$ is symmetric). Recall that $t_{\varphi_1,\nu}$ is missing iff $\nu(\varphi_1)$ evaluates to false in which case $\pi[\![\varphi_1]\!]_\nu = 0$. Then we get $p = p_2 = 0 + p_2 = \pi[\![\varphi_1]\!]_\nu + \pi[\![\varphi_2]\!]_\nu$. Formula $\varphi_1 \vee \varphi_2$ evaluates to false or undetermined if both φ_i are either undetermined or false. In both cases,

the provenance graph will contain both g_1 and g_2 and $p = \pi[\![\varphi]\!]_\nu$.

$\varphi := \exists x\, \varphi_1$: We have $\pi[\![\varphi]\!]_\nu = \sum_{a \in A} \pi[\![\varphi_1]\!]_{\nu[x \mapsto a]}$. There is one rule deriving Q_φ which bounds x, the variable bound by the existential quantifier, to Dom (all values from A) and φ_1. Tuple $t_{\varphi,\nu}$ exists iff there exists at least one $a \in A$ for which $\nu[x \mapsto a](\varphi_1)$ is true which based on the induction hypothesis implies that tuple $t_{\varphi_1,\nu[x\mapsto a]}$ exists. Recall that an existing tuple is connected to successful and undetermined rule derivations. Let $p_a = p_{\varphi_1,\nu[x\mapsto a]}$. Since the tuple node $t_{\varphi,\nu}$ will be replaced with $+$ by $\text{Tr}_{\text{Expl}\to\mathbb{N}[X,\bar{X}]}$, the expression p is a sum $p = \sum_{a \in A \wedge \nu[x\mapsto a](\text{FREE}(\varphi_1))} p_a$. By the induction hypothesis $p_a = \pi[\![\varphi_1]\!]_{\nu[x\mapsto a]}$. Furthermore, if $\nu[x \mapsto a](\text{FREE}(\varphi_1))$ is false then $\pi[\![\varphi_1]\!]_{\nu[x\mapsto a]} = 0$. Thus, $p = \pi[\![\varphi]\!]_\nu$. Tuple $t_{\varphi,\nu}$ is missing if all its derivations fail and undetermined if no derivation is successful and at least one if undetermined. In both cases the provenance graph connects this tuple to all possible derivations and, thus, $p = \pi[\![\varphi]\!]_\nu$.

$\varphi := \forall x\, \varphi_1$: We have $\pi[\![\varphi]\!]_\nu = \prod_{a \in A} \pi[\![\varphi_1]\!]_{\nu[x\mapsto a]}$. The translation uses a standard way of encoding universal quantification in Datalog as double negation ($\neg\, \exists x \neg\, \varphi_1$). Note that there is a single rule deriving Q_φ with a single negated atom $Q_{\varphi'}$ in its body. Let $t' = Q_{\varphi'}(\nu(\text{FREE}(\varphi)))$, g' be the subgraph rooted at t', and $p' = \text{Tr}_{\text{Expl}\to\mathbb{N}[X,\bar{X}]}(g')$. Furthermore, let $t_a = t_{\varphi_1,\nu[x\mapsto a]}$ and $p_a = p_{\varphi_1,\nu[x\mapsto a]}$. Based on step 6 of the translation, t' is replaced by $\cdot$ and get an expressions $p' = \prod_{a \in A'} p_a$ for some $A' \subseteq A$. Now the existence of t' and of the individual t_a determine the set A' (i.e., all t_a that are connected to t' in the provenance graph). Tuple $t_{\varphi,\nu}$ exists (is missing, is undetermined) if t' is missing (exists, is undetermined). We have $p_{\varphi,\nu} = p'$. Now, the body of the rule deriving $Q_{\varphi'}$ consists of one negated atom $\neg\, Q_{\varphi_1}(\text{FREE}(\varphi_1))$. Thus, t' exists if for at least one $a \in A$ we have that $\nu[x \mapsto a](\varphi_1)$ is false, i.e., the universal quantification evaluates to false. In this case t' is connected to all tuples t_a which are existing or undetermined. If we can prove that for every $a \in (A - A')$ we have $\pi[\![\varphi_1]\!]_{\nu[x\mapsto a]} = 0$, then $\pi[\![\varphi]\!]_\nu = p' = p$. Since for every $a \in (A - A')$ we know that t_a is missing, by the induction hypothesis we have $\pi[\![\varphi_1]\!]_{\nu[x\mapsto a]} = 0$. If t' is missing, then all its derivations are failed and all tuples t_a exist. In this case $A = A'$ and $\pi[\![\varphi]\!]_\nu = p' = p$. Finally, t' is undetermined if all its derivations are either failed or undetermined and at least one derivation is undetermined. This is the case when all tuples t_a are either existing or undetermined and at least one t_a is undetermined. In this, case t' is still connected to all tuples t_a and we have again $A = A'$.

$\varphi := x\, \mathbf{op}\, y$: The translation creates a single rule consisting of the comparison and atoms $Dom(x)$ and $Dom(y)$ which ensure safety. There is a single rule derivation for each valuation ν that assigns constants to x and y. We have $\pi[\![x\, \mathbf{op}\, y]\!]_\nu = 1$ if $\nu(x)\, \mathbf{op}\, \nu(y)$ and $\pi[\![x\, \mathbf{op}\, y]\!]_\nu = 0$ otherwise. Tuple $t = Q_\varphi(\nu(x), \nu(y))$ exists if $\nu(x)\, \mathbf{op}\, \nu(y)$. In this case $p_{\varphi,\nu} = 1 \cdot 1 \cdot 1 = 1 = \pi[\![x\, \mathbf{op}\, y]\!]_\nu$. Otherwise, $p_{\varphi,\nu} = 1 \cdot 1 \cdot 0 = \pi[\![x\, \mathbf{op}\, y]\!]_\nu$. □

8 Computing Explanations

We now present our approach for computing explanations using Datalog. Our approach generates a Datalog program $\mathbb{GP}_{P,\psi}$ by rewriting a given query (input program) P to return the edge relation of the explanation $\text{Expl}(P, \psi, I)$ for a provenance question (PQ) ψ. Recall that a PQ is a pattern describing existing/missing outputs of interest and that an explanation for a PQ is a subgraph of the provenance which contains the provenance of all tuples described by the pattern.

Our approach for computing $\mathbb{GP}_{P,\psi}$ consists of the following steps that we describe in detail in the following subsections: 1) we unify the input program P with the PQ ψ by propagating constants from ψ top-down to prune derivations of outputs that do not match the PQ; 2) we determine for each IDB predicate whether the explanation may contain existing, missing, or both types of tuples from this predicate. Similarly, for each rule we determine whether successful, failed, all, or no derivations of this rule may occur in the provenance graph; 3) based on restricted and annotated version of the input program produced by the first two steps, we then generate *firing rules* which capture the variable bindings of successful and failed derivations of the input program's rules; 4) The result of the firing rules is a superset of the set of relevant provenance fragments. We introduce additional rules that enforce connectivity to remove spurious fragments; 5) finally, we create rules that generate the edge relation of the explanation. This is the only step that depends on what provenance type (e.g., Fig. 4) is requested.

In the following, we will illustrate our approach using the provenance question $\psi_{n,s} = \text{Why}\, \mathtt{Q}(n, s)$ from Example 1, i.e., why New York is connected to Seattle via train with one intermediate stop, but there is no direct connection.

8.1 Unifying the Program with the PQ

The node $\mathtt{Q}(n, s)$ in the provenance graph (Fig. 2) is only connected to derivations which return $\mathtt{Q}(n, s)$. For instance, if variable X is bound to another city x (e.g.,

Algorithm 1 Unify Program With PQ

```
1: procedure UnifyProgram(P, ψ))
2:   Q(t) ← Pattern(ψ)
3:   todo ← [Q(t)]
4:   done ← {}
5:   P_Unified = []
6:   while todo ≠ [] do
7:     a ← pop(todo)
8:     insert(done, a)
9:     rules ← getRulesForAtom(P, a)
10:    for all r ∈ rules do
11:      unRule ← unifyRule(r, a)
12:      P_Unified ← P_Unified :: unRule
13:      for all g ∈ body(unRule) do
14:        if g ∉ done then todo ← todo :: g
15:  return P_Unified
```

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 process corresponds to selection push-down for relational algebra expressions. We may create multiple partially unified versions of a rule or predicate. 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 ($\mathtt{T}(n, s)$). We store variable bindings as superscripts to distinguish multiple copies of a rule generated based on different bindings.

Example 9. *Given the question $\psi_{n,s}$, we unify the single rule r_1 using the assignment $(X{=}n, Y{=}s)$:*

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

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 pseudocode for the algorithm is presented in Algorithm 1). For an input P, we use $P_{Unified}$ to denote the result of this unification.

8.2 Add Annotations based on Success/Failure

For Why $Q(t)$ (Whynot $Q(t)$), we are only interested in subgraphs of the provenance rooted at existing (missing) tuple nodes for $\mathtt{Q}$. With this information, we can infer restrictions for the success/failure state of nodes in the provenance graph that are directly or indirectly 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 and use these annotations to guide the generation of rules that capture derivations in step 3. Here, T (F) indicates that we are only interested in successful (failed) nodes, and F/T that we are interested in both.

Example 10. *Continuing with our running example question $\psi_{n,s}$, we know that $\mathtt{Q}(n, s)$ 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 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 (e.g., $\neg\, \mathtt{T}(n, s)^T$). For instance, if $\mathtt{T}$ would be an IDB predicate, then 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. We annotate such atoms with F/T. The algorithm generating the annotation consists of the steps shown below (the pseudocode is presented in Algorithm 2). We use P_{Annot} to denote the result of this algorithm for $P_{Unified}$ (input to this step).

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 positive goal in the rule body, we propagate its annotation (F, T, or F/T) to all rules that have this atom in the head. For negated goals, we propagate the inverted annotation (e.g., F for T) unless the annotation is F/T in which case we propagate F/T.

8.3 Creating Firing Rules

To compute the relevant subgraph of $\mathcal{PG}(P, I)$ (the explanation) for a PQ, we need to determine successful and/or failed rule derivations. Each derivation paired with the information whether it is successful over the

Algorithm 2 Success/Failure Annotations

```
1: procedure AnnotProgram(P_Unified, ψ)
2:   state ← typeof(ψ)
3:   Q(t) ← Pattern(ψ)
4:   todo ← [Q(t)^state]
5:   done ← {}
6:   P_Annot = []
7:   while todo ≠ [] do
8:     a ← pop(todo)
9:     state ← typeof(a)
10:    insert(done, a)
11:    rules ← getRulesForAtom(P_Unified, a)
12:    for all r ∈ rules do
13:      annotRule ← annotRule(r, state)
14:      P_Annot ← P_Annot :: annotRule
15:      for all g ∈ body(annotRule) do
16:        if state = F then
17:          state ← F/T
18:        if isNegated(g) then
19:          state ← switchState(state)
20:        if g^state ∉ done ∧ isIDB(g) then
21:          todo ← todo :: g^state
22:  for all r ∈ P_Annot do
23:    if typeof(r) = F/T then
24:      P_Annot ← removeAnnotatedRules(P_Annot, r, {F, T})
25:  return P_Annot
```

given database (and which goals are failed in case it is not successful) is sufficient for generating a fragment of $\mathcal{PG}(P, I)$. Successful 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. The rationale behind this is that rule derivations apply a valuation to assign constants to the variables of a rule. A rule derivation is successful if after applying the valuation to the body, all goals are successful. For any successful rule derivation, the result of applying the valuation to the head is part of the result of the program containing the rule. Thus, if for a rule r we create a firing rule that has the same body as r, but has all body variables in the head, then the IDB predicate computed by the firing rule stores all successful derivations of r. 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 firing version of the rules r deriving the predicate in the body. These rules capture existing IDB tuples. 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. These rules create copies of EDB relations. Note that for the positive case these rules are not strictly necessary, but as we will show they are necessary once negation and missing answers are introduced.

$$\mathtt{F}_{\mathtt{Q},T}(n,s) :- \mathtt{F}_{\mathtt{r}_1,T}(n,s,Z)$$
$$\mathtt{F}_{\mathtt{r}_1,T}(n,s,Z) :- \mathtt{F}_{\mathtt{T},T}(n,Z), \mathtt{F}_{\mathtt{T},T}(Z,s), \mathtt{F}_{\mathtt{T},F}(n,s)$$
$$\mathtt{F}_{\mathtt{T},T}(n,Z) :- \mathtt{T}(n,Z)$$
$$\mathtt{F}_{\mathtt{T},T}(Z,s) :- \mathtt{T}(Z,s)$$
$$\mathtt{F}_{\mathtt{T},F}(n,s) :- \neg\,\mathtt{T}(n,s)$$

Fig. 8: Example firing rules for Why $\mathtt{Q}(n,s)$

Example 11. *Consider the annotated program in Example 10 for the question* $\psi_{n,s} =$ Why $\mathtt{Q}(n,s)$. *We generate the firing rules shown in Fig. 8. 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* $\mathtt{F}_{\mathtt{r}_1,T}$*, and replacing each goal with its firing version (e.g.,* $\mathtt{F}_{\mathtt{T},T}$ *for the two positive goals and* $\mathtt{F}_{\mathtt{T},F}$ *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* $\mathtt{F}_{\mathtt{T},F}(n,s)$*). In the following, we will define rules with annotation* F *to capture missing tuples of an EDB or IDB predicate and failed rule derivations. Furthermore, we introduce firing rules for EDB tuples (three rules at the bottom). Evaluated over the example instance from Fig. 1 these rules produce the following instance:*

$\mathtt{F}_{\mathtt{Q},T}(n,s)$

$\mathtt{F}_{\mathtt{r}_1,T}(n,s,c)$ $\quad$ $\mathtt{F}_{\mathtt{r}_1,T}(n,s,w)$

$\mathtt{F}_{\mathtt{T},T}(n,c)$ $\quad$ $\mathtt{F}_{\mathtt{T},T}(n,w)$
$\mathtt{F}_{\mathtt{T},T}(c,s)$ $\quad$ $\mathtt{F}_{\mathtt{T},T}(s,s)$ $\quad$ $\mathtt{F}_{\mathtt{T},T}(w,s)$

$\mathtt{F}_{\mathtt{T},F}(n,s)$

We, now, 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 with this head exist (otherwise, we may capture irrelevant failed derivations of existing tuples). In addition, we need to determine which goals are failed for a 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 FT firing rule for a predicate R captures all tuples in Tup(R) no matter whether they exist or not. An additional boolean attribute is used to store for each such tuple whether it exists or not.

Example 12. *Consider an EDB relation* $\mathtt{R}(\mathtt{A},\mathtt{B})$*, domain assignment* $dom(\mathtt{R.A}) = dom(\mathtt{R.B}) = \{a,b\}$ *and instance* $\{\mathtt{R}(a,a),\mathtt{R}(b,b)\}$*. The firing rules for* $\mathtt{F}_{\mathtt{R},F/T}$ *using the queries* $dom_{\mathtt{R.A}}$ *and* $dom_{\mathtt{R.B}}$ *provided by the user to compute the domain assignment are:*

$$\begin{aligned}
\mathtt{F}_{\mathtt{R},F/T}(X,Y,true) &:\!- \mathtt{F}_{\mathtt{R},T}(X,Y)\\
\mathtt{F}_{\mathtt{R},F/T}(X,Y,false) &:\!- \mathtt{F}_{\mathtt{R},F}(X,Y)\\
\mathtt{F}_{\mathtt{R},T}(X,Y) &:\!- \mathtt{R}(X,Y)\\
\mathtt{F}_{\mathtt{R},F}(X,Y) &:\!- dom_{\mathtt{R.A}}(X), dom_{\mathtt{R.B}}(X), \neg\mathtt{R}(X,Y)
\end{aligned}$$

The third rule computes existing tuples creating a copy of relation R*. The fourth rule, enumerates all tuples in* Tup(R) *using the domain assignment and only returns tuples that do not exist. The first and the second rule then combine the results of the last two rules and store whether a tuple exists as a boolean value in an additional attribute. Evaluating these rules we generate the following instance:*

$$\begin{aligned}
&\mathtt{F}_{\mathtt{R},F/T}(a,a,true) && \mathtt{F}_{\mathtt{R},F/T}(b,b,true)\\
&\mathtt{F}_{\mathtt{R},F/T}(a,b,false) && \mathtt{F}_{\mathtt{R},F/T}(b,a,false)
\end{aligned}$$

A firing rule capturing missing tuples may not be safe, i.e., it may contain variables that only occur in negated goals. These variables should be restricted to the associated domains for the attributes the variables are bound to. Recall that associated domain $dom(\mathtt{R.A})$ for an attribute R.A is given as an unary query $dom_{\mathtt{R.A}}$. We use these queries in firing rules to restrict the values a variable is bound to. Thus, we ensure that only missing answers formed from the associated domains are considered and that firing rules are safe.

Example 13. *Consider the question* Whynot $\mathtt{Q}(s,n)$ *from Example 1. The firing rules generated for this question are in Fig. 9. 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* $\mathtt{T}(s,Z)$*. New York cannot be reached from Seattle with exactly one transfer, i.e.,* $\mathtt{Q}(s,n)$ *is not in the result. Thus, we are only interested in failed derivations of rule* r_1 *with* $X{=}s$ *and* $Y{=}n$*. Furthermore, each rule node in the provenance graph corresponding to such a derivation will only be connected to failed subgoals. Thus, we need to capture which goals are successful or failed for each such failed derivation. We model this using boolean variables* V_1*,* V_2*, and* V_3 *(one for each goal) that are set to true iff the tuple corresponding to the goal exists. The firing version* $\mathtt{F}_{\mathtt{r}_1,F}(s,n,Z,V_1,V_2,\neg V_3)$ *of* r_1 *returns all variable bindings for derivations of* r_1 *such that* $\mathtt{Q}(s,n)$ *is the head (i.e., guaranteed by adding* $\mathtt{F}_{\mathtt{Q},F}(s,n)$ *to the body), the rule derivations are failed, and the tuple corresponding to the* i^{th} *goal exists for this binding iff* V_i *is true. The failure status of the* i^{th} *goal is* V_i *for positive goals and* $\neg V_i$ *for negated goals. To produce all these bindings, we need rules capturing successful and failed tuple nodes for each subgoal of the rule* r_1*. We annotate such rules with* F/T *and use a boolean variable (true or false) to record whether a tuple exists (e.g.,* $\mathtt{F}_{\mathtt{T},F/T}(s,Z,true) :\!- \mathtt{F}_{\mathtt{T},T}(s,Z)$ *is one of these rules). Similarly,* $\mathtt{F}_{\mathtt{T},F/T}(s,n,false)$ *represents the fact that tuple* $\mathtt{T}(s,n)$ *(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 unified EDB atom annotated with* F/T*, we create four rules: one for existing tuples (e.g.,* $\mathtt{F}_{\mathtt{T},T}(s,Z) :\!- \mathtt{T}(s,Z)$*), one for the failure case (e.g.,* $\mathtt{F}_{\mathtt{T},F}(s,Z) :\!- dom_{\mathtt{T.toCity}}(Z), \neg\mathtt{T}(s,Z)$*), and two for the* F/T *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.*

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

Fig. 9: Example firing rules for Whynot $\mathtt{Q}(s,n)$

Algorithm 3 takes as input the program P_{Annot} produced by step 2 and outputs a program P_{Fire} containing firing rules. The pseudocode for the subprocedures is presented in Algorithm 4. The algorithm maintains a queue *todo* of annotated atoms that have to be processed which is initialized with Pattern(ψ), i.e., the provenance question atom. Furthermore, we maintain a set *done* of atoms that have been processed already. Variables *todo*, *done*, and P_{Fire} are global variables that are shared with the subprocedures of this algorithm.

For each atom $R(t)^\sigma$ (line 8) from the queue (here σ is the annotation of the atom, e.g., F), we mark the atom as done (line 9). We then consider two cases: R is an EDB atom or an IDB atom in which case we have to create firing rules for the predicate (relation) and the rules deriving it. The firing rules for EDB predicates check whether the tuples are existing or missing in the relation. These rules allow us to determine the success or failure of goals corresponding EDB predicates in rule derivations. For IDB predicates, we create firing rules that determine their existance based on successful or failed rule derivations captured by firing rules for the rules of the program. Consider a given program P with two rules: 1) $r_1 : \mathtt{Q}(X) :- \mathtt{R}(X,Y), \mathtt{Q}_1(Y)$ and 2) $r_2 : \mathtt{Q}_1(Y) :- \mathtt{S}(Y,Z)$ where $\mathtt{R}$ and $\mathtt{S}$ are EDB relations and $\mathtt{Q}$ and $\mathtt{Q}_1$ are IDB predicates. To capture provenance for the predicate $\mathtt{Q}(X)$, we create firing rules for $\mathtt{R}$ and $\mathtt{S}$ to check existence or absence of tuples matching t in $\mathtt{R}$ and $\mathtt{S}$. Moreover, we also generate firing rules for rules r_1 and r_2 to explain how derivations of $\mathtt{Q}(X)$ through these rules have succeeded or failed. The firing rule for r_1 uses the firing rule for IDB predicate $\mathtt{Q}_1$ which in turn uses the firing rule for r_2 since $head(r_2) = \mathtt{Q}_1$. We describe these two cases in the following.

EDB atoms (line 13). For an EDB atom $R(t)^T$, we use procedure CREATEEDBFIRINGRULE to create one rule $\mathtt{F}_{\mathtt{R},T}(t) :- R(t)$ that returns tuples from relation R that match 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 13.

IDB atoms (lines 13-19). IDB atoms with F or F/T annotations are handled in the same way as EDB atoms with these annotations. If the atom is $R(t)^F$ (line 13), we create a rule with $\neg\, \mathtt{F}_{\mathtt{R},T}(t)$ in the body using the associated domain queries to restrict variable bindings. Similarly, for $R(t)^{F/T}$, the procedure called in line 13 adds two additional rules as shown in Fig. 9 (5^{th} and 6^{th} rule) for EDB atoms. Both types of rules only use the positive firing version for $R(t)$ and domain predicates in their body. Thus, these rules are independent of which rules derive R. Now, for any R, we create positive firing rules that correspond to the derivation of R through one particular rule. For that, we iterate over the annotated versions of all rules deriving R (lines 14+15). For each rule r with head $\mathtt{R}(t)$, we create a rule $\mathtt{F}_{\mathtt{R},T}(t) :- \mathtt{F}_{\mathtt{r},T}(\vec{X})$ where $\vec{X}$ is the concatenation of t with all existential variables from the body of r.

Algorithm 3 Create Firing Rules

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

Rules (line 15-19). 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 $\mathtt{F}_{\mathtt{r},T}(\vec{X})$ where $\vec{X} = vars(r)$ (stored in variable $args$, lines 16+17) 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 $\mathtt{F}_{\mathtt{r},F}(\vec{X}, \vec{V})$ 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 $\mathtt{F}_{\mathtt{R},F}$ 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. 9.

Theorem 4 (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 derivations, and failed derivations for missing answers:*

- $\mathtt{F}_{\mathtt{R},T}(t) \in P_{Fire}(I) \leftrightarrow R(t) \in P(I)$
- $\mathtt{F}_{\mathtt{R},F}(t) \in P_{Fire}(I) \leftrightarrow R(t) \notin P(I)$
- $\mathtt{F}_{\mathtt{r},T}(t) \in P_{Fire}(I) \leftrightarrow r(t) \models P(I)$
- $\mathtt{F}_{\mathtt{r},F}(t,\vec{V}) \in P_{Fire}(I) \leftrightarrow r(t) \not\models P(I) \wedge head(r(t)) \notin P(I)$ *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 4 by induction over the "depth" of a program. 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

Algorithm 4 Create Firing Rules Subprocedures

procedure CreateEDBFiringRule(P_{Fire}, $R(t)^{\sigma}$)
  $[X_1, \ldots, X_n] \leftarrow vars(t)$
  $r_T \leftarrow \mathtt{F}_{\mathtt{R},T}(t) :- R(t)$
  $r_F \leftarrow \mathtt{F}_{\mathtt{R},F}(t) :- dom(X_1), \ldots, dom(X_n), \neg R(t)$
  $r_{F/T-1} \leftarrow \mathtt{F}_{\mathtt{R},F/T}(t, true) :- \mathtt{F}_{\mathtt{R},T}(t)$
  $r_{F/T-2} \leftarrow \mathtt{F}_{\mathtt{R},F/T}(t, false) :- \mathtt{F}_{\mathtt{R},F}(t)$
  if $\sigma = T$ **then**
    $P_{Fire} \leftarrow P_{Fire} :: r_T$
  else if $\sigma = F$ **then**
    $P_{Fire} \leftarrow P_{Fire} :: r_F$
  else
    $P_{Fire} \leftarrow P_{Fire} :: r_T :: r_F :: r_{F/T-1} :: r_{F/T-2}$

procedure CreateIDBNegRule(P_{Fire}, $R(t)^{\sigma}$)
  $[X_1, \ldots, X_n] \leftarrow vars(t)$
  if $\sigma \neq T$ **then**
    $r_{new} \leftarrow \mathtt{F}_{\mathtt{R},F}(t) :- dom(X_1), \ldots, dom(X_n), \neg \mathtt{F}_{\mathtt{R},T}(t)$
    $P_{Fire} \leftarrow P_{Fire} :: r_{new}$
  if $\sigma = F/T$ **then**
    $r_T \leftarrow \mathtt{F}_{\mathtt{R},F/T}(t, true) :- \mathtt{F}_{\mathtt{R},T}(t)$
    $r_F \leftarrow \mathtt{F}_{\mathtt{R},F/T}(t, false) :- \mathtt{F}_{\mathtt{R},F}(t)$
    $P_{Fire} \leftarrow P_{Fire} :: r_T :: r_F$

procedure CreateIDBPosRule(P_{Fire}, $R(t)^{\sigma}$, r, $args$)
  $r_{pred} \leftarrow \mathtt{F}_{\mathtt{R},T}(t) :- \mathtt{F}_{\mathtt{r},T}(args)$
  $P_{Fire} \leftarrow P_{Fire} :: r_{pred}$

procedure CreateIDBFiringRule(P_{Fire}, $R(t)^{\sigma}$, r, $args$)
  $body_{new} \leftarrow []$
  for all $g_i(\vec{X}) \in body(r)$ **do**
    $\sigma_{goal} \leftarrow T$
    if isNegated(g_i) **then**
      $\sigma_{goal} \leftarrow F$
    $g_{new} \leftarrow \mathtt{F}_{\mathtt{pred(g_i)},\sigma_{goal}}(\vec{X})$
    $body_{new} \leftarrow body_{new} :: g_{new}$
    if $g_i(\vec{X})^T \notin done \wedge \sigma = T$ **then**
      $todo \leftarrow todo :: g_i(\vec{X})^{\sigma_{goal}}$
  $r_{new} \leftarrow \mathtt{F}_{\mathtt{r},T}(args) :- body_{new}$
  $P_{Fire} \leftarrow P_{Fire} :: r_{new}$
  if $\sigma \neq T$ **then**
    $body_{new} \leftarrow []$
    for all $g_i(\vec{X}) \in body(r)$ **do**
      $g_{new} \leftarrow \mathtt{F}_{\mathtt{pred(g_i)},F/T}(\vec{X}, V_i)$
      $body_{new} \leftarrow body_{new} :: g_{new}$
      if $g_i(\vec{X})^{F/T} \notin done$ **then**
        $todo \leftarrow todo :: g_i(\vec{X})^{\sigma_{goal}}$
      if isNegated(g_i) **then**
        $args \leftarrow args :: \neg V_i$
      else
        $args \leftarrow args :: V_i$
    $r_{new} \leftarrow \mathtt{F}_{\mathtt{r},\sigma}(args) :- body_{new}$
    $P_{Fire} \leftarrow P_{Fire} :: r_{new}$

Algorithm 5 Add Connectivity Joins

procedure AddConnectivityRules(P_{Fire}, ψ)
  $P_{FC} \leftarrow []$
  $Q(t) \leftarrow$ Pattern(ψ)
  $paths \leftarrow$ pathStartingIn($P_{Fire}, Q(t)$)
  for all $p \in paths$ **do**
    for all $e = (r_i(\vec{X_1})^{\sigma_1}, r_j(\vec{X_2})^{\sigma_2}) \in p$ **do**
      $goals \leftarrow$ getMatchingGoals(e)
      for all $g_k \in goals$ **do**
        $g_{new} \leftarrow$ unifyHead($\mathtt{F}_{\mathtt{r_i},\sigma_1}(t_1), g_k, \mathtt{F}_{\mathtt{r_j},\sigma_2}(t_2)$)
        $r_{new} \leftarrow \mathtt{FC}_{\mathtt{r_j},\mathtt{r_i^k},\sigma_2}(t_2) :- body(\mathtt{F}_{\mathtt{r_j},\sigma_2}(t_2)), g_{new}$
        $P_{FC} \leftarrow P_{FC} :: r_{new}$
  return P_{FC}

$$\mathtt{F}_{\mathtt{Q},T}(n,s) :- \mathtt{F}_{\mathtt{r}_1,T}(n,s,Z)$$
$$\mathtt{F}_{\mathtt{r}_1,T}(n,s,Z) :- \mathtt{F}_{\mathtt{T},T}(n,Z), \mathtt{F}_{\mathtt{T},T}(Z,s), \mathtt{F}_{\mathtt{T},F}(n,s)$$
$$\mathtt{FC}_{\mathtt{r}_2,\mathtt{r}_1^1,\mathtt{T}}(n,Z) :- \mathtt{T}(n,Z), \mathtt{F}_{\mathtt{r}_1,T}(n,s,Z)$$
$$\mathtt{FC}_{\mathtt{r}_2,\mathtt{r}_1^2,\mathtt{T}}(Z,s) :- \mathtt{T}(Z,s), \mathtt{F}_{\mathtt{r}_1,T}(n,s,Z)$$
$$\mathtt{FC}_{\mathtt{r}_2,\mathtt{r}_1^3,\mathtt{F}}(n,s) :- \neg\mathtt{T}(n,s), \mathtt{F}_{\mathtt{r}_1,T}(n,s,Z)$$

Fig. 10: Example firing rules with connectivity checks

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 program P has depth 1, e.g., $r : \mathtt{Q}(\vec{X}) :- \mathtt{R}(\vec{X_1}), \ldots, \mathtt{R}(\vec{X_n})$. We first prove that 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 $\mathtt{F}_{\mathtt{R},T}$ creates a copy of the input relation $\mathtt{R}$ and, thus, $\forall t : t \in R \Leftrightarrow t \in \mathtt{F}_{\mathtt{R},T}$. For the negative version $\mathtt{F}_{\mathtt{R},F}$, all variables are bound to associated domains dom and it is explicitly checked that $\neg R(\vec{X})$ holds. Finally, $\mathtt{F}_{\mathtt{R},F/T}$ uses $\mathtt{F}_{\mathtt{R},T}$ and $\mathtt{F}_{\mathtt{R},F}$ to determine whether the tuple exists in $\mathtt{R}$. Since these rules are correct, it follows that $\mathtt{F}_{\mathtt{R},F/T}$ is correct. The positive firing rule for the rule r ($\mathtt{F}_{\mathtt{r},T}$) is correct since its body only contains positive and negative EDB firing rules ($\mathtt{F}_{\mathtt{R},T}$ and $\mathtt{F}_{\mathtt{R},F}$, respectively) which are already known to be correct. The correctness of the positive firing version of a rule's head predicate ($\mathtt{F}_{\mathtt{Q},T}$) follows naturally from the correctness of $\mathtt{F}_{\mathtt{r},T}$. The negative version of the rule $\mathtt{F}_{\mathtt{r},F}(\vec{X}, \vec{V})$ contains an additional goal (i.e., $\neg\mathtt{Q}(\vec{X})$) and uses the firing version $\mathtt{F}_{\mathtt{R},F/T}$ to return only bindings for failed derivations. For a head predicate with annotation F, we create two firing rules ($\mathtt{F}_{\mathtt{Q},T}$ and $\mathtt{F}_{\mathtt{Q},F}$). The rule $\mathtt{F}_{\mathtt{Q},T}$ was already proven to be correct. $\mathtt{F}_{\mathtt{Q},F}$ is also correct, because it contains only $\mathtt{F}_{\mathtt{Q},T}$ and domain queries in the body which were already proven to be correct.

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 $\max_{R \in body(r)} d(R) \leq n$, 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. □

8.4 Connectivity Joins

To be in the result of a firing rule is a necessary, but not sufficient, condition for the corresponding rule node to be connected to a node $Q(t') \in \text{MATCH}(\psi)$ in the explanation. Thus, we have to check connectivity of intermediate results explicitly.

Example 14. *Consider the firing rules for $\psi_{n,s}$ shown in Fig. 8. The corresponding rules with connectivity checks are shown in Fig. 10. All rule nodes corresponding to $\mathtt{F}_{\mathtt{r}_1,T}(n,s,Z)$ are guaranteed to be connected to the node $\mathtt{Q}(n,s)$ (corresponding to the only atom in $\text{MATCH}(\psi_{n,s})$). Note that connectivity joins are also required for negative firing rules (e.g., $\mathtt{F}_{\mathtt{r}_1,F}(s,n,Z,V_1,V_2,\neg V_3)$ in Fig. 9 is used for* WHYNOT *). For sake of example, assume that instead of using* T*, rule r_1 uses an IDB relation R which is computed using a rule $r_2 : \mathtt{R}(X,Y) :- \mathtt{T}(X,Y)$. Consider the firing rule $\mathtt{F}_{\mathtt{r}_2,T}(n,Z) :- \mathtt{T}(n,Z)$ created based on the 1^{st} goal of r_1. Some provenance 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 successful binding of r_1. That is, for the node $\mathtt{R}(n,c)$, there has to exist a tuple $\mathtt{R}(c,s)$. Connectivity is achieved by adding the head of the firing rule for r_1 to the body of the firing rule for r_2 as shown in Fig. 10 (the 3^{rd} and 4^{th} rule).*

Our algorithm traverses the query's rules starting from PQ atom(s) 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 r_i's body. 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 $\mathtt{FC}_{\mathtt{r}_j,\mathtt{r}_i^k,\mathtt{T}}(\vec{X})$ 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 add the unified head of r_i to the body of r_j. These rules check whether rule nodes in the provenance graph are connected to nodes in $\text{MATCH}(\psi)$.

8.5 Computing the Edge Relation

The program created so far captures sufficient information for generating the edge relation of the explanation for a PQ (which is used when rendering graphs). We make this step part of the program to offload this work to database backend. To compute the edge relation, we use Skolem functions to create node identifiers. An identifier records the type of the node (tuple, rule, or goal), variables assignments, and the success/failure status of the node, e.g., a tuple node $\mathtt{T}(n,s)$ that is successful would be represented as $f^T_{\mathtt{T}}(n,s)$. Each rule firing corresponds to a fragment of $\mathcal{PG}(P,I)$. For example, one

Algorithm 6 Create Edge Relation

```
1:  procedure CreateEdgeRelation(P_FC, ψ)
2:    P_M ← []
3:    Q(t) ← Pattern(ψ)
4:    todo ← [Q(t)]
5:    done ← {}
6:    while todo ≠ [] do
7:      R(t)^σ ← pop(todo)
8:      done ← insert(done, R(t)^σ)
9:      rules ← getRules(R(t)^σ)
10:     for all r ∈ rules do
11:       args ← args(head(r))
12:       if isEDB(R) then
13:         if σ = T then
14:           if isNegated(g) then
15:             r_{g→R} ← edge(f_g^T(t), f_R^F(t)) :- F_{r,T}(args)
16:           else
17:             r_{g→R} ← edge(f_g^T(t), f_R^T(t)) :- F_{r,T}(args)
18:         else
19:           if isNegated(g) then
20:             r_{g→R} ← edge(f_g^F(t), f_R^T(t)) :- F_{r,F}(args)
21:           else
22:             r_{g→R} ← edge(f_g^F(t), f_R^F(t)) :- F_{r,F}(args)
23:         P_M ← P_M :: r_{g→R}
24:       else
25:         r_new ← edge(f^σ_{pred(r)}(t), f_r^{σ_r}(t,...)) :- F_{r,σ}(t,...)
26:         P_M ← P_M :: r_new
27:         for all g(t) ∈ body(r) do
28:           if isNegated(g) then
29:             σ' ← switchState(σ)
30:           else
31:             σ' ← σ
32:           todo ← todo :: g(t)^{σ'}
33:           if σ' = T then
34:             r_{r→g} ← edge(f_r^T(args), f_g^T(t)) :- F_{r,T}(args)
35:           else
36:             r_{r→g} ← edge(f_r^F(args), f_g^F(t)) :- F_{r,F}(args)
37:           P_M ← P_M :: r_{r→g}
38:   return P_M
```

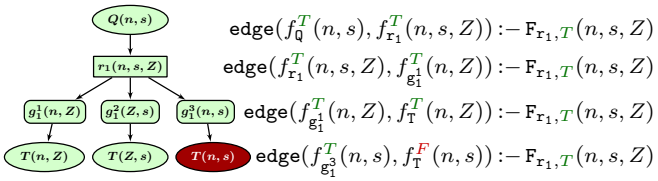


Fig. 11: Fragment of an explanation corresponding to a derivation of rule r_1 (left) and the rules generating the edge relation for such a fragment (right)

such fragment is shown in Fig. 11 (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 (only failed goals for failed rules)
- One rule creating edges between each goal node and the corresponding EDB tuple node

Algorithm 7 Create Lineage Edge Relation

```
1: procedure CreateLinEdgeRelation(P_FC, ψ)
2:   P_M ← []
3:   Q(t) ← Pattern(ψ)
4:   todo ← [Q(t)]
5:   done ← {}
6:   while todo ≠ [] do
7:     R(t)^σ ← pop(todo)
8:     if R(t)^σ ∈ done then
9:       continue
10:    done ← insert(done, R(t)^σ)
11:    rules ← getRules(R(t)^σ)
12:    for all r ∈ rules do
13:      args ← args(head(r))
14:      if σ = T then
15:        if isNegated(g) then
16:          r_{Q→R} ← edge(f_Q^T(t), f_R^F(t)) :- F_{r,T}(args)
17:        else
18:          r_{Q→R} ← edge(f_Q^T(t), f_R^T(t)) :- F_{r,T}(args)
19:      else
20:        if isNegated(g) then
21:          r_{Q→R} ← edge(f_Q^F(t), f_R^T(t)) :- F_{r,F}(args)
22:        else
23:          r_{Q→R} ← edge(f_Q^F(t), f_R^F(t)) :- F_{r,F}(args)
24:      P_M ← P_M :: r_{Q→R}
25:  return P_M
```

Example 15. *Consider the firing rules with connectivity joins from Example 14. Some of the rules for creating the edge relation of the explanation sought by the user are shown in Fig. 11 (right). 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, and the* 2^{nd} *rule creates an edge between* $r_1(n,s,Z)$ *and* $g_1^1(n,Z)$*. Edges for failed derivations are created by considering the corresponding node identifiers and a failure pattern (e.g.,* $\mathtt{F}_{\mathtt{r}_1,F}(s,n,Z,V_1,V_2,\neg V_3)$*).*

8.6 $\mathcal{K}$-Explanations

To compute one of the $\mathcal{K}$-explanation types introduced in Sec. 6.2, we only have to adapt the rules generating the edge relation. As an example, we present the modifications for computing $\textsc{Expl}_{\mathsf{Which}(X)}$ (e.g., Fig. 4c). Recall that semiring $\mathsf{Which}(X)$ models provenance as a set of contributing tuples and we encode this as a graph by connecting a head of a rule derivation to the atoms in its body. That is, for the $\textsc{Expl}_{\mathsf{Which}(X)}$, we create only one type of rule that connects tuple nodes for the head predicate to EDB tuple nodes. We use $\mathbb{GP}_{P,\psi}^{\mathsf{Which}(X)}$ to denote the program generated in this way for an input program P, and a PQ ψ.

Example 16. *Consider the graph fragment for* r_1 *in Fig. 11 (left) without rule and goal nodes. The rule that creates the edge between* $\mathtt{Q}(n,s)$ *and* $\mathtt{T}(n,Z)$ *is*

$$\mathtt{edge}(f_{\mathtt{Q}}^{T}(n,s), f_{\mathtt{T}}^{T}(n,Z)) :- \mathtt{F}_{\mathtt{r}_1,T}(n,s,Z)$$

For each successful derivation of result $\mathtt{Q}(n,s)$ *using rule* r_1*, a subgraph replacing* Z *with bindings from the derivation is included in* $\textsc{Expl}_{\mathsf{Which}(X)}$*.*

8.7 Correctness

We now prove that our approach is correct.

Theorem 5. *Let* P *be a program,* I *be a instance, and* ψ *a* PQ*. Program* $\mathbb{GP}_{P,\psi}$ *evaluated over* I *returns the edge relation of* $\textsc{Expl}(P,\psi,I)$*.*

Proof. To prove Theorem 5, we have to show that 1) only edges from $\mathcal{PG}(P,I)$ are in $\mathbb{GP}_{P,\psi}(I)$ and 2) the program returns precisely the set of edges of explanation $\textsc{Expl}(P,\psi,I)$. **1.** The constants used as variable binding by the rules creating edges in $\mathbb{GP}_{P,\psi}$ are either constants that occur in the PQ ψ 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 $\mathbb{GP}_{P,\psi}$ is strictly based on the structure of the input program and connects nodes that agree on variable bindings. Thus, each edge produced by $\mathbb{GP}_{P,\psi}$ will be contained in $\mathcal{PG}(P,I)$.
2. We now prove that the program $\mathbb{GP}_{P,\psi}$ returns precisely the set of edges of $\textsc{Expl}(P,\psi,I)$. Assume that the PQ ψ only uses constants (the extension to PQs which contain variables is immediate). Consider a rule of an input program of depth 1 (i.e., only EDB predicates in the rule body). For such a rule node to be connected to an atom $Q(t) \in \textsc{Match}(\psi)$, its head variables have to be bound to t (guaranteed by the unification step in Sec. 8.1). 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. 8.4). 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 $n+1$. □

Theorem 6. *Let P be a positive program, I be a database instance, and ψ a* PQ. *The result of program $\mathbb{GP}^{\mathsf{Which}(X)}_{P,\psi}$ is the edge relation of* $\textsc{Expl}_{\mathsf{Which}(X)}(P, \psi, I)$.

Proof. We prove Theorem 6 by induction over the structure of a program as in the proof of Theorem 4. **1) Base Case.** Consider a program P with depth 1 and P has a single IDB predicate $\mathtt{Q}$, i.e., only containing rules of the form $r_i : \mathtt{Q}(\vec{X}) :- \mathtt{R}^{\mathtt{i}}_1(\vec{X^i_1}), \ldots, \mathtt{R}^{\mathtt{i}}_{\mathtt{n_i}}(\vec{X^i_{n_i}})$, where each $\mathtt{R_j}$ is an EDB relation and all goals are positive. According to [15], the semiring annotation of a tuple $\mathtt{Q}(t)$ in the result of a positive datalog program is computed as a sum of products. This sum contains one monomial per successful rule derivation with head $\mathtt{Q}(t)$. Such a monomial is constructed by multiplying the annotations of the grounded goals in the rule derivation. Let ν denote a variable assignment corresponding to a rule derivation and $Val(r, \mathtt{Q}(t))$ the set of all variable assignment for the rule r that yield $\mathtt{Q}(t)$. Since addition and multiplication are idempotent in $\mathsf{Which}(X)$, the annotation of a result $\mathtt{Q}(t)$ is computed as below:

$$\bigcup_{i} \bigcup_{\nu \in Val(r_i, \mathtt{Q}(t))} \bigcup_{j=1}^{n_i} \nu(R^i_j(X^i_j))$$

That is, the $\mathsf{Which}(X)$ expression of $\mathtt{Q}(t)$ contains the set of annotations of all tuples that appear in at least one successful derivation of $\mathtt{Q}(t)$. In $\textsc{Expl}_{\mathsf{Which}(X)}$, the fact that a tuple $\mathtt{R}(t')$ belongs to the provenance of $\mathtt{Q}(t)$ is recorded as an edge from $\mathtt{Q}(t)$ to $\mathtt{R}(t')$. Thus, to prove that the generated $\textsc{Expl}_{\mathsf{Which}(X)}$ graph correctly encodes $\mathsf{Which}(X)$, we have to show that such an edge exists for every goal of a successful rule derivation with $\mathtt{Q}(t)$ as the head. In Theorem 4, we have proven that firing rules correctly determine existence of tuples and successful/failed derivations for the user's provenance question. The adapted algorithm creates one rule for every goal of a rule which returns an edge if the goal is part of a successful rule derivation (this is ensured by using firing rules). Thus, an edge exists for every goal of a successful derivation of $\mathtt{Q}(t)$.

2) Inductive Step. Assume that the algorithm is correct for any program of depth less than n. Consider the program P with depth n and a derivation of a rule r of depth n in this program. Based on the induction hypothesis, we know that the $\mathsf{Which}(X)$ annotation for each atom in the body of the derivation is recorded correctly. From Theorem 4 and using the same argument as in the base case, the rules created for rule r will generate edges that link the result tuple of rule r to each atom in its body (the claim holds). □

9 Factorization

For provenance polynomials, we can exploit the distributivity law of semirings to generate factorizations of provenance [31] which are exponentially more concise in the best case. For instance, consider a query r_3 returning the end points of paths of length 2 evaluated over the edge-labelled graph in Fig. 12a. The provenance polynomial for the query result $\mathtt{Q_{2hop}}(d)$ using the annotations from Fig. 12a is shown in Fig. 12d. Each monomial in the polynomial corresponds to one of the derivations of the result using r_3. Each of these $2 \cdot (2^2)$ (we have two options as starting points and, for each hop, we have two options) derivations corresponds to one path of length 2 ending in d. When generating provenance graphs for provenance polynomials, we create "$\cdot$" nodes for rule derivations and "$+$" nodes for IDB tuples. Fig. 12b is the factorized representation of this polynomial. We can exploit the fact that our approach shares common subexpressions to produce a particular factorization. This is achieved by rewriting the input program to partition a query by materializing joins and projections as new IDB relations which can then be shared. We first review f-trees and d-trees as introduced in [32] which encode possible nesting "schemas" for factorized representations of provenance (or query results), the size bounds for factorized representations based on d-trees proven in [32], and how to chose a d-tree for a query that results in the optimal worst-case size bound for the factorized representation of the provenance according to this d-tree. Then, we introduce a query transformation for conjunctive queries which, given an input query and the d-tree for this query, generates a rewritten query which returns a provenance graph factorized corresponding to this d-tree. We employ this rewriting to produce more concise provenance in PUG (experiments are shown in Sec. 11).

Factorized Representations. In [31,32], a *factorized representation* (f-rep for short) of a relation is defined as an algebraic expression constructed using singleton relations (one tuple with one value) and the relational operators union and product. Any f-rep over a set of attributes from a schema S can be interpreted as a relation over S by evaluating the algebraic expression, e.g., $\{(a)\} \times (\{(b)\} \cup \{(c)\})$ is a factorized representation of the relation $\{(a, b), (a, c)\}$. Following the convention from [31], we denote a singleton $\{(a)\}$ as a. Factorization can be applied to compactly represent relations and

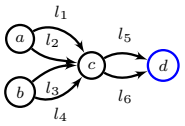
$r_3 : \mathtt{Q_{2hop}}(X) :- \mathtt{H}(Y, L_1, Z), \mathtt{H}(Z, L_2, X)$

$r_4 : \mathtt{Q_{2hop-d}}() :- \mathtt{H}(Y, L_1, Z), \mathtt{H}(Z, L_2, d)$

$r_5 : \mathtt{Q_{2hop}}() :- \mathtt{Q}_{L_1}(Z), \mathtt{Q}_{L_2}(Z)$

$r_{5'} : \mathtt{Q}_{L_1}(Z) :- \mathtt{H}(Y, L_1, Z)$

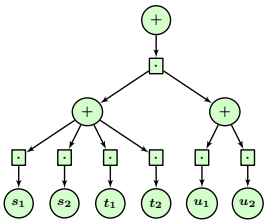
$r_{5''} : \mathtt{Q}_{L_2}(Z) :- \mathtt{H}(Z, L_2, d)$



Relation H

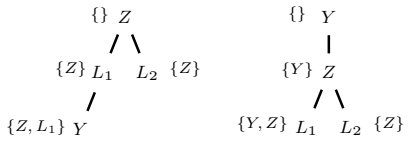
S	L	E	
a	l_1	c	s_1
a	l_2	c	s_2
b	l_3	c	t_1
b	l_4	c	t_2
c	l_5	d	u_1
c	l_6	d	u_2

(a) 2hop queries (r_3 and r_4), rewriting (r_5, r'_5, r''_5) according to d-tree $\mathcal{T}_1$, and example database (graph)

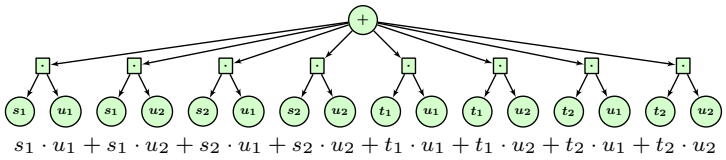


$(s_1 + s_2 + t_1 + t_2) \cdot (u_1 + u_2)$

(b) Factorized representation (r_5, $r_{5'}$, $r_{5''}$)



(c) Two d-trees of r_4: $\mathcal{T}_1$ (left) and $\mathcal{T}_2$ (right)



(d) Flat representation (r_4)

Fig. 12: Factorized and flat provenance graphs ($\mathbb{N}[X]$) explaining WHY $\mathtt{Q_{2hop}}(d)$ and two d-trees for r_4.

query results as well as provenance (e.g., Fig. 12b). We will factorize representations of provenance which encode variables of provenance polynomials as the tuples annotated by these variables and show how to extract provenance polynomials from provenance graphs generated in this way.

F-trees for F-reps. Olteanu et al. [32] introduce *f-trees* to encode the nesting structure of f-reps. At first, let us consider only f-trees which encode the nesting structure of a boolean query [31]. An f-tree for a boolean query Q (e.g., r_4 in Fig. 12a) is a rooted forest with one node for every variable of Q.[4] An f-rep according to an f-tree $\mathcal{T}$ nests values according to $\mathcal{T}$: a node labelled with X corresponds to a union of values from the attributes bound to X by the query. The values of attributes bound to children of a node X corresponding to a single value x bound to X are grouped under x. If a node has multiple children, then their f-reps are connected via $\times$. For example, consider an f-tree $\mathcal{T}$ with root X and a single child Y for a query $\mathtt{Q}() :- \mathtt{R}(X, Y)$. An f-rep of $\mathtt{Q}$ according to $\mathcal{T}$ would be of the form $x_1 \times (y_{1_1} \cup \ldots \cup y_{n_1}) \cup \ldots \cup x_m \times (y_{1_m} \cup \ldots \cup y_{n_m})$, i.e., the Y values co-occurring with a given X value x are grouped as a union and then paired with x. An f-tree encodes (conditional) independence of the variables of a query in the sense that the values of one variable do not depend on the values of another variable. For instance, two siblings X and Y in an f-tree have to be independent since a union of X values is paired (cross-product) with a union of Y values. This is only correct if the values of X and Y are independent. The independence assumptions encoded in an f-tree may not hold for every possible query with the same schema as the f-tree. Thus, only some f-trees with a particular schema may be applicable for a query with this schema. It was shown in [32], that a query has an f-rep over an f-tree $\mathcal{T}$ for any database iff for each relation in Q the variables assigned to attributes of this relation (these variables are called dependent) are on the same root-to-leaf path in the f-tree. This is called the *path condition*. Note that multiple references to the same relation in a query are considered as separate relations when checking this condition. For instance, consider the boolean query r_4 in Fig. 12a which checks if there are paths of length 2 ending in the node d. Fig. 12c shows two f-trees $\mathcal{T}_1$ and $\mathcal{T}_2$ for this query (ignore the sets on the side of nodes for now). An f-rep according to $\mathcal{T}_2$ for r_4 would encode a union of Y values paired ($\times$) with a union of Z values for this Y value. Each Z value nested under a Y value is then paired with a cross-product of L_1 and L_2 values.

D-trees for D-reps. The size of a factorized representation can be further reduced by allowing subexpressions to be shared through definitions, i.e., using algebra graphs instead of trees. In [32], such representations are called *d-representations* (d-rep). Analogous to how f-trees define the structure of f-reps, d-trees were introduced to define the structure of d-reps. A d-tree is an f-tree where each node X is annotated with a set $key(X)$, a subset of its ancestors in the f-tree on which the node and any of its dependents depend on. The f-rep of the subtree rooted in X is unique for each combination of values from $key(X)$. That is, if $key(X)$ is a strict subset of the ancestors of X, then the same d-rep for the subtree at X can be shared by multiple ancestors, reducing the size of the representation. In Fig. 12c, the set key is shown beside each node, e.g., in $\mathcal{T}_2$, the variable L_2 depends only on Z, but not on Y. An important result proven in [32] is that, for a given d-tree $\mathcal{T}$ for a query Q, the size of d-rep of Q over a database

[4] In [32], relational algebra is used to express queries and nodes of f-trees represent equivalence classes of attributes which in Datalog correspond to query variables.

I is bound by $|I|^{s^\uparrow(\mathcal{T})}$ where $s^\uparrow(\mathcal{T})$ is a rational number computed based on $\mathcal{T}$ alone (see [32] for details of how to compute $s^\uparrow(\mathcal{T})$). This bound can be used to determine the d-tree for a query Q which will yield the d-rep of worst-case optimal size by enumerating the valid d-trees for Q and, then, chosing the one with the lowest value of $s^\uparrow$.

Example 17. *Consider the d-rep for r_4 (Fig. 12a) over the example instance of relation* H *(Fig. 12a) according to d-tree $\mathcal{T}_2$ (Fig. 12c). Variable Y at the root of $\mathcal{T}_2$ is bound to the attribute S from the first reference of* H*, i.e., the starting point of paths of length 2 ending in d. There are two such starting points a and b. Now each of these are paired with the only valid intermediate node c on these paths (variable Z). Finally, for this node, we compute the cross-product of the L_1 and L_2 values connected to c. Since the L_2 values only depend on Z, we share these values when the same Z value is paired with multiple Y values. The final result is $(a \times c \times (l_1 \cup l_2) \times l^\uparrow) + (b \times c \times (l_3 \cup l_4) \times l^\uparrow)$ where $l^\uparrow := (l_5 \cup l_6)$.*

Factorization of Provenance. For the provenance of a conjunctive query Q that is not a boolean query, i.e., it has one or more variables in the head (e.g., r_3 in Fig. 12a), we have to compute a provenance polynomial for each result of Q. We would like the factorization of the provenance of Q to clearly associate the provenance polynomial of a result t with the tuple t. That is, we want to avoid factorizations where head variables of Q are nested below variables that store provenance (appear only in the body) since reconstructing the provenance polynomial for t would require enumeration of the full provenance from the factorized representation in the worst case. For example, consider a query with head variable X and body variable Y. If Y is the root of a d-tree $\mathcal{T}$, then the d-rep of Q according to $\mathcal{T}$ would be of the form $y_1 \times (x_{1_1} + \ldots + x_{n_1}) + \ldots + y_m \times (x_{1_m} + \ldots + x_{n_m})$. To extract the provenance polynomial for a result x_i, we may have to traverse all y values since there is no indication, for which y values, x_i appears in the sum $x_{1_i} + \ldots + x_{n_i}$. We ensure this by constructing d-trees which do not include the head variables, but treat those as ancestors of every node in the d-tree when computing *key* for the nodes. For instance, to make $\mathcal{T}_1$ (Fig. 12c) a valid d-tree for capturing the provenance of r_3 (Fig. 12a), we treat the head variable X as a virtual ancestor of all nodes and get $key(Z) = \{X\}$ and $key(L_2) = \{Z, X\}$. Furthermore, if we are computing an explanation to a provenance question (PQ) ψ that binds one or more head variables to constants, then we can propagate these bindings before constructing a d-tree for the query. For example, to explain $\mathtt{Q_{2hop}}(d)$, we would propagate the binding $X = d$ resulting in rule r_4 (Fig. 12a). Thus, any d-tree for r_4 can be used to create a factorized $\textsc{Expl}_{\mathbb{N}[X]}$ graph for the user question $\textsc{Why}(\mathtt{Q_{2hop}}(d))$.

Rewriting Queries for Factorization. We now explain how, given a d-tree $\mathcal{T}$ for a conjunctive query Q and positive PQ $\psi := \textsc{Why}\, Q(t)$, to generate a Datalog query Q_{rewr} such that, for any database I, we have that $\textsc{Expl}_{\mathbb{N}[X]}(Q_{rewr}, \psi, I)$ encodes $\mathbb{N}[X](Q_{rewr}, I, t)$ for each $t \in \textsc{Match}(\psi)$ factorized according to $\mathcal{T}$. We first unify the query with the PQ as described in Sec. 8.1. Given a unified input query Q and a d-tree $\mathcal{T}$, we compute Q_{rewr} as follows.

1. Assume a total order among the variables of Q (e.g., the lexicographical order). For every node X with children $Y_1, \ldots, Y_n$ in the d-tree $\mathcal{T}$, we generate
$$r_X : \mathtt{Q_X}(key(X)) :- \mathtt{Q_{Y_1}}(key(Y_1)), \ldots, \mathtt{Q_{Y_n}}(key(Y_n))$$
2. Now for every atom $\mathtt{R}(Z_1, \ldots, Z_m)$ in the body of Q, we find the shortest path starting in a root node that contains all nodes Z_1 to Z_m. Let $Y = Z_i$ for some i be the last node on this path. Then, we add atom $\mathtt{R}(Z_1, \ldots, Z_m)$ to the body of rule r_Y created in the previous step.
3. Let $X_1, \ldots, X_n$ be the roots of the d-tree $\mathcal{T}$ (being a forest, a d-tree may have multiple roots). Furthermore, let $Y_1, \ldots, Y_m$ denote the head variables of the unified input query Q with the PQ. We create
$$r_Q : \mathtt{Q}(Y_1, \ldots, Y_m) :- \mathtt{Q_{X_1}}(key(X_1)), \ldots, \mathtt{Q_{X_n}}(key(X_n))$$

The rewriting above creates a factorization according to a d-tree $\mathcal{T}$. However, it may contain rules which cannot potentially lead to reuse and, thus, result in overhead that could be avoided if we were able to identify such rules. We now present an optimization that removes such rules to further reduce the size of the generated provenance graphs. Consider two nodes X and Y in a d-tree where Y is the only child of X, i.e., $key(Y) = key(X) \cup \{X\}$. We would generate rules

$$r_X : \mathtt{Q_X}(key(X)) :- \mathtt{Q_Y}(X \cup key(X))$$
$$r_Y : \mathtt{Q_Y}(X \cup key(X)) :- \ldots$$

In this case, the intermediate result Q_Y does not lead to further factorization (we have a union of unions). Thus, we can merge the rules by substituting the atom $\mathtt{Q_Y}(X \cup key(X))$ in r_X with the body of r_Y. A similar situation may arise with the rule r_Q deriving the final query result. In general, we can merge any rule of the form $\mathtt{Q_1}(X_1, \ldots, X_n) :- \mathtt{Q_2}(X_1, \ldots, X_n)$ with the rule deriving $\mathtt{Q_2}$ (in our translation, there will be exactly one rule with head $\mathtt{Q_2}$).

Example 18. *Consider the question* $\text{WHY}\,\mathtt{Q}_{\mathtt{2hop}}(d)$ *over the query* r_3 *from Fig. 12a. Unifying the query with this question yields* r_4 *(below* r_3 *in the same figure). To rewrite the query according to the d-tree* $\mathcal{T}_1$ *from Fig. 12c, we apply the above algorithm to create rules:*

$$r_{Q_{2hop}} : \mathtt{Q}_{\mathtt{2hop}}() :- \mathtt{Q}_{\mathtt{Z}}() \qquad r_Z : \mathtt{Q}_{\mathtt{Z}}() :- \mathtt{Q}_{\mathtt{L}_1}(Z), \mathtt{Q}_{\mathtt{L}_2}(Z)$$
$$r_{L_1} : \mathtt{Q}_{\mathtt{L}_1}(Z) :- \mathtt{Q}_{\mathtt{Y}}(Z, L_1) \qquad r_Y : \mathtt{Q}_{\mathtt{Y}}(Z, L_1) :- \mathtt{H}(Y, L_1, Z)$$
$$r_{L_2} : \mathtt{Q}_{\mathtt{L}_2}(Z) :- \mathtt{H}(Z, L_2, d)$$

Applying the optimizations introduced above, we merge the rules $r_{Q_{2hop}}$ *with* r_Z *(the head* $\mathtt{Q}_Z$ *is the body of* $r_{Q_{2hop}}$*). Since* $key(Y) = key(L_1) \cup \{L_1\}$ *and* L_1 *has only one child, we merge* r_Y *into* r_{L_1}*. The resulting program is shown as rules* r_5*,* $r_{5'}$ *and* $r_{5''}$ *in Fig. 12a.*

Factorized Explanations. To generate a concise factorization of provenance for a PQ ψ over a conjunctive query Q, we first find a d-tree $\mathcal{T}$ with minimal $s^{\uparrow}$ among all d-trees for Q (such a d-tree $\mathcal{T}$ guarantees worst-case optimal size bounds for the generated factorization). Then, we rewrite the input query according to $\mathcal{T}$ (explained above) and use the approach in Sec. 8 to generate $\text{EXPL}_{\mathbb{N}[X]}(Q_{rewr}, \psi, I)$ encoding the d-rep of $\mathbb{N}[X](Q_{rewr}, I, t)$ for each $t \in \text{MATCH}(\psi)$.

Example 19. *Continuing with Example 18, assume we compute the* $\mathbb{N}[X]$ *explanation using the rewritten query (*r_5*,* $r_{5'}$*, and* $r_{5''}$*). The result over the example database is shown in Fig. 12b. The top-most addition and multiplication correspond to the successful derivation using rule* r_5 *(using* c *as an intermediate hop from some node to* d*). The left branch below the multiplication encodes the four possible derivations of* $\mathtt{Q}_{\mathtt{L}_1}(c)$ *(*$s_1 + s_2 + t_1 + t_2$*) and the right branch corresponds to the two derivations of* $\mathtt{Q}_{\mathtt{L}_2}(c)$ *(*$u_1 + u_2$*). The polynomial captured by this graph is* $(s_1 + s_2 + t_1 + t_2) \cdot (u_1 + u_2)$*. That is, there are 4 ways to reach* c *from any starting node and two ways of reaching* d *from* c *leading to a total of* $4 \cdot 2 = 8$ *paths of length two ending in the node* d*.*

10 Implementation

We have implemented the approach presented in this paper in a system called *PUG* (Provenance Unification through Graphs). PUG is an extension of GProM [1], a middleware that executes provenance requests using a relational database backend (shown in Fig. 13). 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

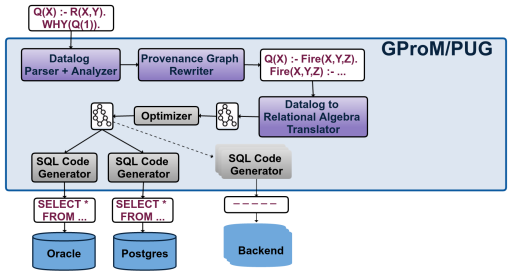


Fig. 13: PUG implementation in GProM

predicate 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. 8 are applied to create the program $\mathbb{GP}_{P,\psi}$ for the input program P and the provenance question ψ which computes $\text{EXPL}(P, \psi, I)$ (analogously, $\text{EXPL}_{\mathcal{K}}(P, \psi, I)$ for $\mathbb{GP}^{\mathcal{K}}_{P,\psi}$). This program is then translated into relational algebra ($\mathcal{RA}$). The resulting algebra expression is translated into SQL and sent to the backend database to compute the edge relation of the explanation for the question. Based on this edge relation, we render a provenance graph. For examples and installation guidelines see: `https://github.com/IITDBGroup/PUG`. 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 $\mathcal{RA}$ expressions provided by GProM [30] and use its library of $\mathcal{RA}$ to SQL translators. Our translation of first-order queries (a program with a distinguished answer relation) to $\mathcal{RA}$ is mostly standard. See [25] for details and an example.

11 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 benchmark dataset (`http://www.tpc.org/tpch/default.asp`). We mainly evaluate three aspects; 1) we compare our approach for computing explanations (EXPL) with the approach introduced for provenance games [24]. We call the provenance game approach `Direct Method (DM)`, because it directly constructs the full provenance graph; 2) we compare our approach for Lineage ($\text{EXPL}_{\mathsf{Which}(X)}$) to the language-integrated approach developed for the *Links* programming language [9]; 3) we evaluate the performance impact of rewriting queries to produce factorized provenance (Sec. 9). We have created subsets of the DBLP dataset with 100, 1K, 10K, 100K, 1M, and 8M co-author pairs (tuples). For the TPC-H bench-

r_1 : only2hop(X,Y) :– DBLP(X,Z), DBLP(Z,Y), ¬ DBLP(X,Y)

r_2 : XwithYnotZ(X,Y) :– DBLP(X,Y), ¬ $Q_1(X)$

$r_{2'}$: $Q_1(X)$:– DBLP$(X,$'Svein Johannessen'$)$

r_3 : only3hop(X,Y) :– DBLP(X,A), DBLP(A,B), DBLP(B,Y), ¬ $E_1(X)$, ¬ $E_2(X)$

$r_{3'}$: $E_1(X)$:– DBLP(X,Y)

$r_{3''}$: $E_2(X)$:– DBLP(X,A), DBLP(A,Y)

r_4 : ordPriority(X,Y) :– CUSTOMER(A,X,B,C,D,E,F,G), ORDERS(H,A,I,J,K,Y,M,N,O)

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

r_6 : partNotAsia(X) :– PART(A,X,B,C,D,E,F,G,H), PARTSUPP(A,I,J,K,L), SUPPLIER(I,M,N,O,P,Q,R), NATION(O,S,T,U), ¬ $R_1(T,$'ASIA'$)$

$r_{6'}$: $R_1(T,Z)$:– REGION(T,Z,V)

r_7 : suppCust(N) :– SUPPLIER(A,B,C,N,D,E,F), CUSTOMER(G,H,I,N,J,K,L,M)

Fig. 14: DBLP and TPC-H queries for experiments

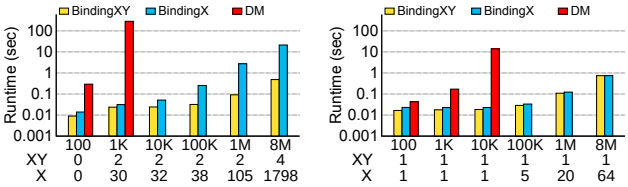
mark, we used database sizes 10MB, 100MB, 1GB, and 10GB. All experiments were run on a machine with 2 x 3.3Ghz AMD Opteron 4238 CPUs (12 cores in total) and 128GB RAM running Oracle Linux 6.4. We use the commercial DBMS X (name omitted due to licensing restrictions) and Postgres as a backend (default is DBMS X). Unless stated otherwise, each experiment was repeated 100 times (we stopped executions that ran longer than 10 minutes) and we report the median runtime. Computations that did not finish within the allocated time are omitted from the graphs.

Workloads. We compute explanations for the queries in Fig. 14. For DBLP datasets, we consider: `only2hop` (r_1) which is our running example query in this paper; `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 (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; `suppCust` (r_7) returns nations having both suppliers and customers.

Implementing DM. 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 compute a conservative lower bound for the runtime of the step constructing the game graph by executing a query (n-way cross-product over the active domain). Note that the actual runtime will

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	-	-	-
TPC-H (Size)	**10MB**	**100MB**	**1GB**	**10GB**
> 10 Variables (r_4, r_5, r_6, r_7)	-	-	-	-

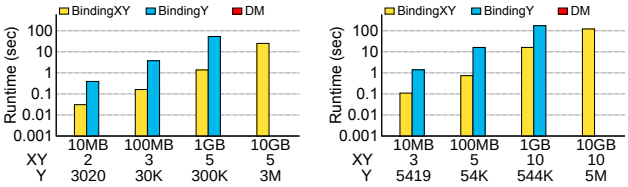
Fig. 15: Runtime of DM in seconds. For entries with '-', the computation did not finish within 10 min.



(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

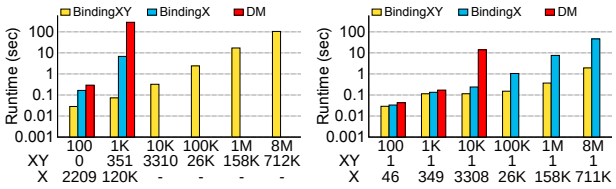
Query \ Binding	X	Y
(d) ordPriority	Customer16	1-URGENT
(e) ordDisc	Customer16	0

(f) Variable bindings for TPC-H PQs

Fig. 16: Why questions: DBLP (top), TPC-H (bottom)

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. 15. 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 of generating explanations for why questions over the queries r_1, r_2, r_4, and r_5 (Fig. 14) is shown in Fig. 16. For the evaluation, we consider the effect of different binding patterns on performance. Fig. 16c and 16f show which variables are bound by the provenance questions (PQs). Fig. 16a and 16b show the runtime for DBLP queries r_1 and r_2, respectively. We also provide the number of rule nodes in the explanation 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,

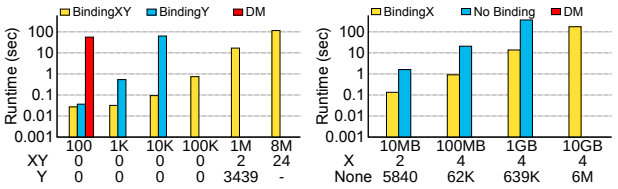


(a) Runtime of `only2hop` (b) Runtime of `XwithYnotZ`

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

(c) Variable bindings for DBLP PQs

Fig. 17: Why-not questions over the DBLP dataset



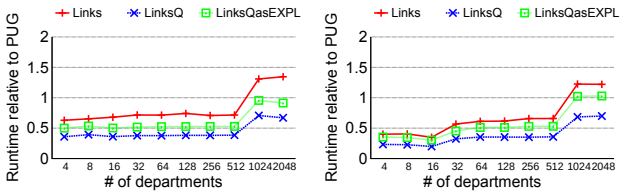
(a) `only3hop` (DBLP) (b) `partNotAsia` (TPC-H)

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. 18: Why questions for queries with negation



(a) `Q7` (employee) (b) `QF3` (employee)

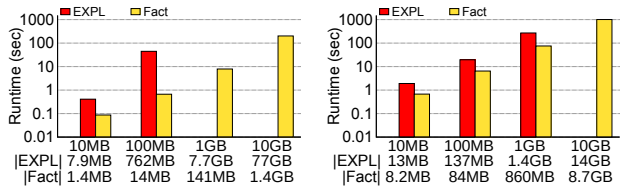
Fig. 19: Comparing Which(X) in PUG with Links

the explanation for `only2hop` with `BindingX` explains why persons are indirect, but not direct, co-authors of "Tore Risch". If both X and Y are bound (`BindingXY`), then the provenance for r_1 and r_2 is limited to a particular indirect and direct co-author, respectively. The runtime for generating explanations grows roughly linear in the dataset size and outperforms `DM` even for small instances. Furthermore, Fig. 16d and 16e (for r_4 and r_5, respectively) show that our approach can handle queries with many variables (attributes in TPC-H) 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 PQ 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 use queries r_1 and r_2 from Fig. 14 to evaluate the performance of computing explanations for failed derivations. When binding all variables in the PQ (`BindingXY`) using the bindings from Fig. 17c, these queries check if a particular set of authors do not 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 is a direct co-author of "Svein Johannessen". The results for queries r_1 and r_2 (DBLP) are shown in Fig. 17a and 17b, respectively. The number of tuples produced by the provenance computation (the number of rule nodes is shown below the X axis) is quadratic in the database size resulting in a quadratic increase in runtime. `DM` only finishes within the allocated time for very small datasets while our approach scales to larger instances.

Queries with Negation. Recall that our approach also handles queries with negation. We choose rules r_3 (multiple negated goals) and r_6 (one negated goal) from Fig. 14 to evaluate the performance of answering why questions over such queries. We use the bindings shown in Fig. 18c. The results for r_3 and r_6 are shown in Fig. 18a and 18b, respectively. These results demonstrate that our approach efficiently computes explanations for such queries. 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 `partNotAsia` (Fig. 18b), which contains many variables and negation exhibits the same trend as queries that have no negation (i.e., r_4 and r_5 in Fig. 16d and Fig. 16e, respectively).

Comparison with Links. In this experiment, we compare the runtime of computing $\text{Expl}_{\text{Which}(X)}$ (e.g., Fig. 4c) with computation of Lineage in Links$^{\text{L}}$ from [9]. We show relative runtimes where PUG is normalized to 1. For this particular evaluation, we use Postgres as a backend since it is supported by both PUG and Links. Note that $\text{Expl}_{\text{Which}(X)}$ contains a full description of each tuple unlike Links$^{\text{L}}$ which returns tuple identifiers (OIDs in Postgres). To get a nuanced understanding of the system's performance, we show three runtimes for Links: 1) `Links` is the actual implementation in Links which computes Lineage (only OIDs) and where the runtime includes the construction of in-memory Links types from the provenance fetched from Postgres; 2) `LinksQ` is the runtime of the queries that Links uses to capture Lineage; and 3) `LinksQasEXPL` which joins the output of `LinksQ` with the base tables (i.e., as informative as $\text{Expl}_{\text{Which}(X)}$). To implement `LinksQ`, we extract provenance capture queries (in SQL) from Postgres while computing `Links`, and use to explicitly compare the query runtimes with $\text{Expl}_{\text{Which}(X)}$. For gener-



(a) Runtime of `suppCust` (b) Runtime of `ordDisc`

Fig. 20: Explanations vs. factorized explanations

ating `LinksQasEXPL` queries, we take the tuple ids and relation names in `LinksQ`, and use these to join with the corresponding relations in the database to return fully described Lineage as $\text{Expl}_{\text{Which}(X)}$. We choose two queries from [9]. The query Q7 applies a range condition to the result of a two-way join. QF3 is a self-join on equality with an additional inequality condition (see [9] for more details). The queries are expressed over two tables `dept` and `emp`. The number of departments is varied from 4 to 2048 (by powers of 2 to replicate the setting from [9]), and each department has 100 employees on average. The relation `dept` consists of one attribute (department name), and `emp` has three attribute (department name, employee name, and salary). QF3 can be written in Datalog as:

$$\texttt{QF3}(N1, N2) :- \texttt{emp}(\_, D, N1, S), \texttt{emp}(\_, D, N2, S), N1 \neq N2$$

The runtimes of queries Q7 and QF3 are shown in Fig. 19a and 19b, respectively. Links performs better on smaller instances. The gap between Links and PUG shrinks with increasing dataset size. PUG outperforms `Links` and `LinksQasEXPL` on larger datasets.

Factorized Explanations. We now compare the performance of generating provenance for a query (`EXPL`) and a factorized representation of provenance (`Fact`) by rewriting the input query (Sec. 9). Factorization techniques perform best for many-to-many joins (e.g., the query r_7 in Fig. 14). The rewritten version of `suppCust` (r_7) producing factorized provenance is shown below.

$$r_8 : \texttt{suppCust}(N) :- \texttt{supp}(N), \texttt{cust}(N)$$
$$r_{8'} : \texttt{supp}(N) :- \texttt{SUPPLIER}(A, B, C, N, D, E, F)$$
$$r_{8''} : \texttt{cust}(N) :- \texttt{CUSTOMER}(G, H, I, N, J, K, L, M)$$

For this experiments, we use a 15 minute time-out. The runtimes for r_7 (yellow bars) and r_8 (red bars) are shown in Fig. 20a. We show the total result size in bytes below the X axis. The runtime of `Fact` grows roughly linear unlike `EXPL` whose growth is quadratic in dataset size. We also evaluate query r_5 which includes one-to-many joins to see how `Fact` performs for a query (Fig. 20b) where factorization only reduces size by a constant factor. This is confirmed by the measurements: the performance of `Fact` for r_5 is $\sim 30\%$ that of `EXPL` independent of dataset size.

12 Conclusions

We present a provenance model and unified framework for explaining answers and non-answers over first-order queries expressed in Datalog. Our efficient middleware implementation generates a Datalog program that computes the explanation for a provenance question and compiles this program into SQL. We prove that our model is expressive enough to encode a wide range of provenance models from the literature and extend our approach to produce concise, factorized representations of provenance. In future work, we will investigate summarization of provenance (we did present a proof-of-concept in [26]) to deal with the large size of explanations for missing answers. We plan to also support query-based explanations [2,3,4,38] and more expressive query languages (e.g., aggregation).

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. N. Bidoit, M. Herschel, and K. Tzompanaki. Immutably answering why-not questions for equivalent conjunctive queries. In *TaPP*, 2014.
3. N. Bidoit, M. Herschel, K. Tzompanaki, et al. Query-Based Why-Not Provenance with NedExplain. In *EDBT*, pages 145–156, 2014.
4. A. Chapman and H. V. Jagadish. Why Not? In *SIGMOD*, pages 523–534, 2009.
5. J. Cheney, L. Chiticariu, and W. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.
6. C. V. Damásio, A. Analyti, and G. Antoniou. Justifications for logic programming. In *Logic Programming and Nonmonotonic Reasoning*, pages 530–542, 2013.
7. D. Deutch, A. Gilad, and Y. Moskovitch. Selective provenance for datalog programs using top-k queries. *PVLDB*, 8(12):1394–1405, 2015.
8. D. Deutch, T. Milo, S. Roy, and V. Tannen. Circuits for datalog provenance. In *ICDT*, pages 201–212, 2014.
9. S. Fehrenbach and J. Cheney. Language-integrated provenance. *Science of Computer Programming*, 2017.
10. J. Flum, M. Kubierschky, and B. Ludäscher. Total and partial well-founded datalog coincide. In *ICDT*, pages 113–124, 1997.
11. B. Glavic, S. Köhler, S. Riddle, and B. Ludäscher. Towards constraint-based explanations for answers and non-answers. In *TaPP*, 2015.
12. 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. Springer, 2013.
13. E. Grädel and V. Tannen. Semiring provenance for first-order model checking. *arXiv preprint arXiv:1712.01980*, 2017.
14. T. Green. Containment of conjunctive queries on annotated relations. *Theory of Computing Systems*, 49(2):429–459, 2011.
15. T. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

16. 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.
17. T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update Exchange with Mappings and Provenance. In *VLDB*, pages 675–686, 2007.
18. T. J. Green and V. Tannen. The semiring framework for database provenance. In *PODS*, pages 93–99, 2017.
19. M. Herschel, R. Diestelkämper, and H. B. Lahmar. A survey on provenance: What for? what form? what from? *The VLDB Journal*, pages 1–26, 2017.
20. M. Herschel and M. Hernandez. Explaining Missing Answers to SPJUA Queries. *PVLDB*, 3(1):185–196, 2010.
21. 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.
22. G. Karvounarakis and T. J. Green. Semiring-annotated data: queries and provenance. *SIGMOD Record*, 41(3):5–14, 2012.
23. 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.
24. 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. Springer, 2013.
25. S. Lee, S. Köhler, B. Ludäscher, and B. Glavic. A SQL-middleware unifying why and why-not provenance for first-order queries. In *ICDE*, pages 485–496, 2017.
26. S. Lee, X. Niu, B. Ludäscher, and B. Glavic. Integrating Approximate Summarization with Provenance Capture. In *TaPP*, 2017.
27. 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.
28. A. Meliou, W. Gatterbauer, and D. Suciu. Reverse data management. *PVLDB*, 4(12), 2011.
29. A. Meliou and D. Suciu. Tiresias: The database oracle for how-to queries. In *SIGMOD*, pages 337–348, 2012.
30. X. Niu, R. Kapoor, B. Glavic, D. Gawlick, Z. H. Liu, V. Krishnaswamy, and V. Radhakrishnan. Provenance-aware query optimization. In *ICDE*, pages 473–484, 2017.
31. D. Olteanu and J. Závodnỳ. Factorised representations of query results: Size bounds and readability. In *ICDT*, pages 285–298. ACM, 2012.
32. D. Olteanu and J. Závodnỳ. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)*, 40(1):2, 2015.
33. S. Riddle, S. Köhler, and B. Ludäscher. Towards constraint provenance games. In *TaPP*, 2014.
34. S. Roy, L. Orr, and D. Suciu. Explaining query answers with explanation-ready databases. *Proceedings of the VLDB Endowment*, 9(4):348–359, 2015.
35. S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *SIGMOD*, 2014.
36. P. Senellart. Provenance and probabilities in relational databases. *ACM SIGMOD Record*, 46(4):5–15, 2018.
37. V. Tannen. Provenance analysis for FOL model checking. *ACM SIGLOG News*, 4(1):24–36, 2017.
38. Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *SIGMOD*, pages 15–26, 2010.
39. E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8):553–564, 2013.
40. 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.
41. J. Xu, W. Zhang, A. Alawini, and V. Tannen. Provenance analysis for missing answers and integrity repairs. *Data Engineering*, page 39, 2018.
42. 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.