The VLDB Journal (2019) 28:47–71
https://doi.org/10.1007/s00778-018-0518-5

REGULAR PAPER

PUG: a framework and practical implementation for why and why-not provenance

Seokki Lee[1] · Bertram Ludäscher[2] · Boris Glavic[1]

Received: 7 November 2017 / Revised: 2 August 2018 / Accepted: 7 August 2018 / Published online: 23 August 2018

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.

Keywords Datalog · Provenance · Missing answers · Semirings

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 from [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 [43] 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.

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 $Q(n, s)$) with

✉ Seokki Lee
slee195@hawk.iit.edu

Bertram Ludäscher
ludaesch@illinois.edu

Boris Glavic
bglavic@iit.edu

[1] Illinois Institute of Technology, 10 W 31st Street, Chicago, IL 60616, USA

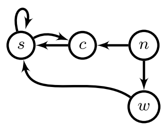
[2] University of Illinois, Urbana-Champaign (UIUC), 501 E. Daniel St, Champaign, IL 61820, USA

[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).

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

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

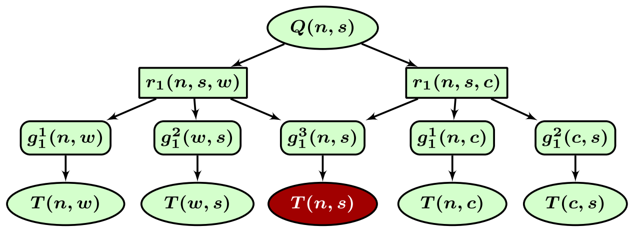


Fig. 2 Provenance graph explaining WHY $\texttt{Q}(n, s)$

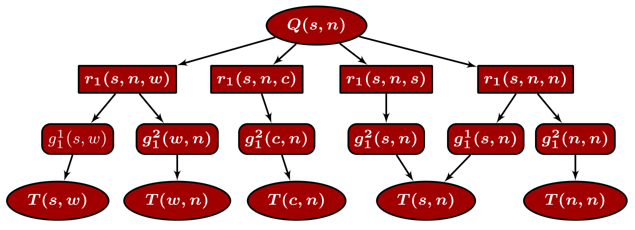


Fig. 3 Provenance graph explaning WHYNOT $\texttt{Q}(s, n)$

one intermediate stop but not directly or why it is not possible to reach New York from Seattle in the same fashion (WHYNOT $\texttt{Q}(s, n)$).

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 $\texttt{Q}(n, s)$ and WHYNOT $\texttt{Q}(s, n)$ are shown in Figs. 2 and 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 $\texttt{Q}(n, s)$. 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 using 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 $\texttt{Q}(n, s)$ based on 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 1st 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 Figure 3 shows the explanation for WHYNOT $\texttt{Q}(s, n)$, i.e., why it is not true that New York is reachable from Seattle with exactly one transfer, but not directly. The tuple $\texttt{Q}(s, n)$ 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 $\texttt{Q}(s, n)$ 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* $\texttt{T}(s, w)$ in red), and Washington DC to New York (*tuple node* $\texttt{T}(w, n)$ in red). The successful goal $\neg\,\texttt{T}(s, n)$ (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 with other provenance models, our model is more syntax-driven. We argue that this 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 [13,39,43]. 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., [30,31,39]). 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 [39].

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 a 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* [26] (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 [33,34].

Contributions This paper extends our previous work [26] 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 [39] 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 Sect. 2 and review Datalog in Sect. 3. We define our model in Sect. 4 and prove it to be equivalent to provenance games in Sect. 5. We, then, show how our approach relates to semiring provenance for positive queries and FO model checking in Sects. 6 and 7, respectively. We present our approach for computing explanations in Sect. 8, and factorization in Sect. 9. Section 10 covers our implementation in *PUG* which we evaluate in Sect. 11. We conclude in Sect. 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 our approach to be equivalent to provenance games in Sect. 5. Köhler et al. [24] presented 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 contrast, 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 Sect. 5), our graphs also encode $\mathbb{N}[X]$ and, thus, all other provenance models expressible as semirings (see Sect. 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 [38]. The semiring model has also been applied to record provenance of model checking for first-order (FO) logic formulas [13,39,43]. 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. [29] computed 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. [36,37] employed 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. [41].

Why-not and missing answers Approaches for explaining missing answers are either based on the query [2–4,40] (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! [42]. 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 [11,35] and provenance graphs [28] 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. [44] leveraged 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 Sect. 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 $\text{TUP}(\mathtt{R})$ to denote the set of all possible tuples for R, i.e., $\text{TUP}(\mathtt{R}) = dom(\mathtt{R.A}_1) \times \ldots \times dom(\mathtt{R.A}_n)$.

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 labeling 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 jth 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 $head(r(\vec{c})) \notin P(I)$. Figures 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 WHY $Q(t)$ or WHYNOT $Q(t)$ where $t = (v_1, \ldots, v_n)$ consists of variables and domain constants ($dom(\mathtt{Q.A})$ for each attribute $\mathtt{Q.A}$). We define:

$$\begin{aligned}
\text{PATTERN}(\psi) &= Q(t)\\
\text{MATCH}(\text{WHY } Q(t)) &= \{Q(t') | t' \in P(I) \wedge t' \preccurlyeq t\}\\
\text{MATCH}(\text{WHYNOT } Q(t)) &= \{Q(t') | t' \notin P(I) \wedge t' \preccurlyeq t \wedge t' \in \text{TUP}(Q)\}
\end{aligned}$$

In Examples 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 $\textsc{Why}\ Q(t)$. 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 $\textsc{Whynot}\ Q(t)$ 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 Definition 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 $\textsc{Tr}_{\Gamma \rightarrow \textsc{Expl}}$ that maps provenance games to graphs and its inverse $\textsc{Tr}_{\textsc{Expl} \rightarrow \Gamma}$. 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 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 labeled L if the corresponding derivation is successful and W otherwise.[2] Given such a solved game (denoted as $\Gamma(P, I)$), we can extract a subgraph rooted at an IDB tuple $\texttt{Q}(t)$ as the provenance of $\texttt{Q}(t)$. Similar to how we derive an explanation for a PQ with $\textsc{Pattern}(\psi) = Q(t)$ 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 $\textsc{Tr}_{\textsc{Expl} \rightarrow \Gamma}$ of a provenance graph into the corresponding game and the reverse transformation $\textsc{Tr}_{\Gamma \rightarrow \textsc{Expl}}$ are straightforward. Thus, we only sketch $\textsc{Tr}_{\textsc{Expl} \rightarrow \Gamma}$ here. EDB tuple nodes are expanded to subgraphs $\neg\texttt{R}(t) \rightarrow \texttt{R}(t) \rightarrow r_R(t)$ for existing tuples and $\neg\texttt{R}(t) \rightarrow \texttt{R}(t)$ for missing tuples. IDB tuple nodes are always expanded to subgraphs of the later form. Rule and goal nodes and their inter-connections are preserved. Goal nodes are connected to negated tuple nodes (positive goals) and to positive tuple nodes (negated goals). For positive tuple and goal

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

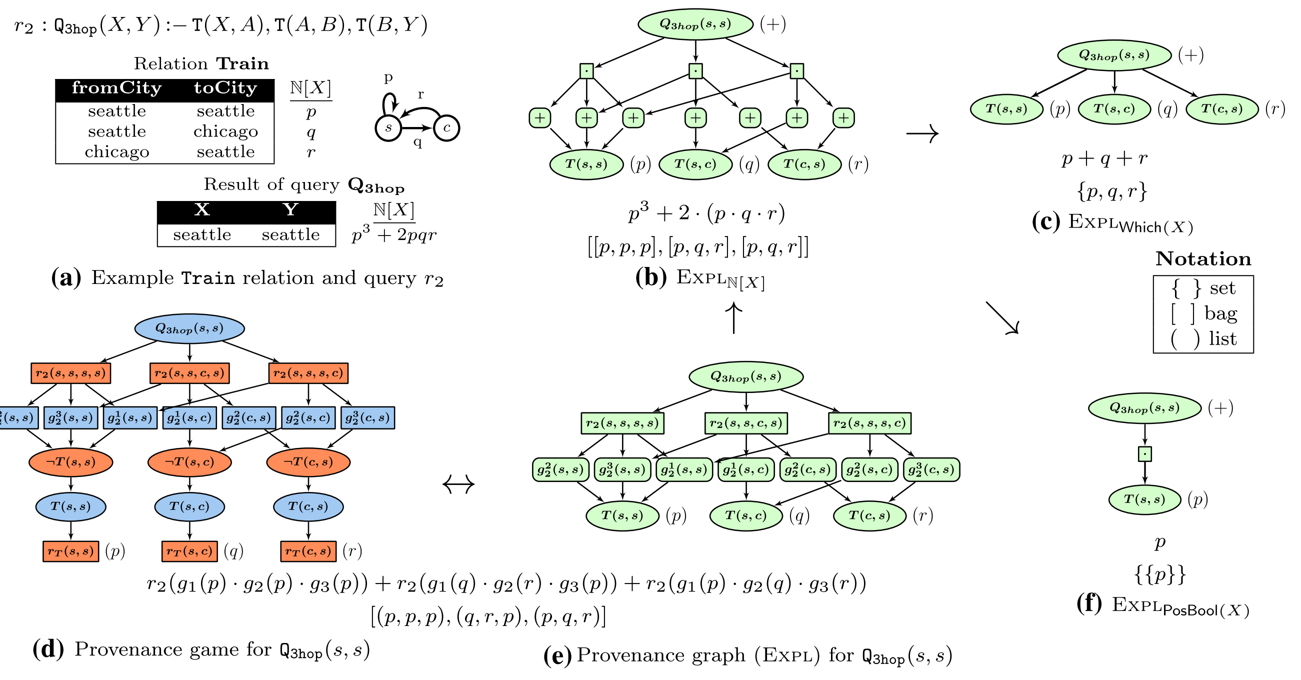

$r_2 : \mathtt{Q_{3hop}}(X,Y) :- \mathtt{T}(X,A), \mathtt{T}(A,B), \mathtt{T}(B,Y)$
Relation Train
fromCity toCity $\mathbb{N}[X]$
seattle seattle p
seattle chicago q
chicago seattle r
Result of query $\mathbf{Q_{3hop}}$
X Y $\mathbb{N}[X]$
seattle seattle $p^3 + 2pqr$
(a) Example Train relation and query r_2
$p^3 + 2 \cdot (p \cdot q \cdot r)$
$[[p,p,p],[p,q,r],[p,q,r]]$
(b) $\textsc{Expl}_{\mathbb{N}[X]}$
$p + q + r$
$\{p, q, r\}$
(c) $\textsc{Expl}_{\mathsf{Which}(X)}$
Notation
{ } set
[ ] bag
( ) list
$r_2(g_1(p) \cdot g_2(p) \cdot g_3(p)) + r_2(g_1(q) \cdot g_2(r) \cdot g_3(p)) + r_2(g_1(p) \cdot g_2(q) \cdot g_3(r))$
$[(p,p,p),(q,r,p),(p,q,r)]$
(d) Provenance game for $\mathtt{Q_{3hop}}(s,s)$
(e) Provenance graph (Expl) for $\mathtt{Q_{3hop}}(s,s)$
p
$\{\{p\}\}$
(f) $\textsc{Expl}_{\mathsf{PosBool}(X)}$

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

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.

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

$$\textsc{Tr}_{\Gamma \to \textsc{Expl}}(\Gamma(P,\psi,I)) = \textsc{Expl}(P,\psi,I)$$
$$\textsc{Tr}_{\textsc{Expl} \to \Gamma}(\textsc{Expl}(P,\psi,I)) = \Gamma(P,\psi,I)$$

Proof See our accompanying technical report [27]. □

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, e, 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 $\neg T(X,Y) \rightarrow T(X,Y) \rightarrow r_T(X,Y)$ 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) $s \rightarrow s \rightarrow s \rightarrow s$, (2) $s \rightarrow c \rightarrow s \rightarrow s$, and (3) $s \rightarrow s \rightarrow c \rightarrow s$.

6 Semiring provenance for positive queries

The semiring annotation model [15,18,22] 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 Sect. 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 and 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 $h : \mathcal{K}_1 \rightarrow \mathcal{K}_2$ 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.

$\underline{(\mathbb{N}[X], +, \cdot, 0, 1)}$: 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 $\mathtt{Q_{3hop}}(s, s)$ 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.

$\underline{(\mathsf{PosBool}(X), +, \cdot, 0, 1)}$: 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 $\mathtt{Q_{3hop}}(s, s)$ is simplified as follows: $p^3 + 2pqr = p + pqr = p$.

$\underline{(\mathsf{Which}(X), +, \cdot, 0, 1)}$: 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 been 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 $\textsc{Tr}_{\textsc{Expl}\rightarrow\Gamma}$.

Theorem 2 *Let* $\textsc{Tr}_{\textsc{Expl}\rightarrow\mathbb{N}[X]}$ *denote the function* $\textsc{Tr}_{\Gamma\rightarrow\mathbb{N}[X]} \circ \textsc{Tr}_{\textsc{Expl}\rightarrow\Gamma}$. *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,*

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

Proof The proof is shown in [27]. □

Consider the explanation for Why $\mathtt{Q_{3hop}}(s, s)$ shown in Fig. 4e. Recall that there are three options for reaching Seattle from Seattle with two intermediate stops corresponding to three derivations of $\mathtt{Q_{3hop}}$ 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., $r_2(s, s, s, s) \rightarrow \cdot$) and (2) replace goal nodes with addition (e.g., $g_2^1(s, s) \rightarrow +$) 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 $\mathtt{Q_{3hop}}(s, s)$ 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 (labeled with $\cdot$). To extract $\mathbb{N}[X]$, we (1) replace labels of leaf nodes with their annotations from Fig. 4a (e.g., $T(s, s)$ is replaced with p) and (2) replace IDB tuple nodes with addition.

Semiring $\mathsf{PosBool}(X)$ $\textsc{Expl}_{\mathsf{PosBool}(X)}$ (Fig. 4f) is computed from $\textsc{Expl}_{\mathbb{N}[X]}$ by first collapsing rule nodes if the subgraphs rooted at these rule nodes are isomorphic and dropping all the goal nodes. Then, "·" 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 "·" node. Applying this process to our example, after the first step, two "·" nodes (one connects $\texttt{Q}_{\texttt{3hop}}(s, s)$ 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 ($\texttt{T}(s, s)$) 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., $\texttt{Q}_{\texttt{3hop}}(s, s)$) with other tuple nodes (e.g., $\texttt{T}(s, s)$) as shown in Fig. 4c.

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

7 Semiring provenance for FO model checking

The semiring framework was recently extended for capturing provenance of first-order (FO) model checking [13,39]. 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 [43]. Another interesting aspect of [13] is that it allows some facts to be left undetermined (their truth is undecided). This enables how to queries [31], 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] and 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 $\textsc{Tl}_{\varphi \to Q}$.

7.1 K-interpretations and dual polynomials

In [18,39], 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 $\neq$).

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

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

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

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

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 [13,39] 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 $R(\mathbf{a}) \wedge \neg R(\mathbf{a}) \equiv \mathit{false}$. 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).

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

$$\pi(R(\mathbf{a})) = x \qquad \pi(\neg R(\mathbf{a})) = \bar{x} \qquad \text{(undetermined)}$$

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 straightforward. 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:

– 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).

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

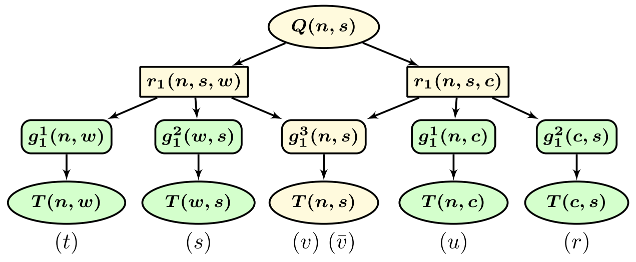


Fig. 5 Provenance graph for WHY $\mathtt{Q}(n, s)$ when $T(n, s)$ is left undetermined

– A rule node is successful (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 throughout the provenance graph.

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

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

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

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

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

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 $\mathrm{TL}_{\varphi \to Q}$ 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 $\textsc{free}(\varphi)$ 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 $\textsc{free}(\varphi) = \{y\}$. Our translation $\mathrm{TL}_{\varphi \to Q}$ 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 subformulas of φ. Here, we assume the existence of a unary predicate *Dom* whose extension is the domain A. Most translation rules are straightforward 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 $\textsc{free}(\varphi)$ 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 $\varphi := \forall x\ \exists y\ \mathrm{R}(x, y)$ checks whether all nodes in the graph have outgoing edges. Let $\varphi_1 = \exists y\ \mathrm{R}(x, y)$ and $\varphi_2 = \mathrm{R}(x, y)$. Translating this formula, we get:

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

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

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 $\mathrm{TL}_{\varphi \to Q}(\varphi)$ rooted at the tuple node $Q_\varphi(\nu(\textsc{free}(\varphi)))$. First, we apply $\mathrm{TL}_{\varphi \to Q}(\varphi)$ to compute Q_φ. Then, we generate an instance I_π where the existence of a tuple corresponding to a literal $\mathrm{R}(\mathbf{a})$ is determined based on the truth value of this literal encoded by its annotation $\pi(\mathrm{R}(\mathbf{a}))$. A tuple $\mathrm{R}(\mathbf{a})$ exists in I_π if $\pi(\mathrm{R}(\mathbf{a})) = x$ or $\pi(\mathrm{R}(\mathbf{a})) = 1$ and $\pi(\neg\, \mathrm{R}(\mathbf{a})) = 0$, the tuple is missing if $\pi(\neg\, \mathrm{R}(\mathbf{a})) = \bar{x}$ or $\pi(\neg\, \mathrm{R}(\mathbf{a})) = 1$ and $\pi(\mathrm{R}(\mathbf{a})) = 0$, and the tuple's existence is undetermined if $\pi(\mathrm{R}(\mathbf{a})) = x$ and $\pi(\neg\, \mathrm{R}(\mathbf{a})) = \bar{x}$. Note that this corresponds to the truth value according to the 5 cases we have discerned in Sect. 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 $Q_\varphi(\nu(\textsc{free}(\varphi)))$, one for each

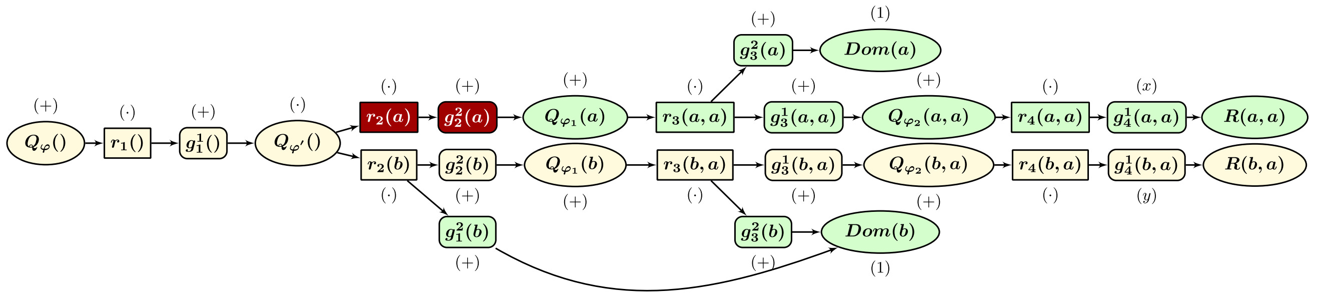


Fig. 7 Provenance graph for query Q_φ based on $\varphi := \forall x\, \exists y\, \mathrm{R}(x, y)$ when $R(a, a)$ is true ($\pi(\mathrm{R}(a, a)) = x$ and $\pi(\neg\,\mathrm{R}(a, a)) = 0$) and $R(b, a)$ is left undetermined ($\pi(\mathrm{R}(b, a)) = y$ and $\pi(\neg\,\mathrm{R}(b, a)) = \bar{y}$).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

valuation ν of the free variables, and the subgraph rooted at one such tuple node encodes $\pi [\![\varphi]\!]_\nu$. By applying a function $\mathrm{Tr}_{\mathrm{EXPL}\rightarrow\mathbb{N}[X,\bar{X}]}$ (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 $\mathrm{Tr}_{\mathrm{EXPL}\rightarrow\mathbb{N}[X,\bar{X}]}$ 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 $\mathrm{Tr}_{\mathrm{EXPL}\rightarrow\mathbb{N}[X,\bar{X}]}$ 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.

$\mathrm{Tr}_{\mathrm{EXPL}\rightarrow\mathbb{N}[X,\bar{X}]}$ 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 $\mathrm{R}(\mathbf{a})$ is replaced by $\pi(\mathrm{R}(\mathbf{a}))$ if the goal is positive and $\pi(\neg\,\mathrm{R}(\mathbf{a}))$ 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 $\mathrm{R}(b, a)$ undetermined, and sets all other positive literals to false without provenance tracking, i.e., $\pi(\mathrm{R}(a, a)) = x$, $\pi(\neg\,\mathrm{R}(a, a)) = 0$, $\pi(\mathrm{R}(b, a)) = y$, $\pi(\neg\,\mathrm{R}(b, a)) = \bar{y}$, and for all other $\mathrm{R}(\mathbf{a})$ we have $\pi(\mathrm{R}(\mathbf{a})) = 0$ and $\pi(\neg\mathrm{R}(\mathbf{a})) = 1$. That is, in I_π, tuple $\mathrm{R}(a, a)$ exists, tuple $\mathrm{R}(b, a)$'s existence is undetermined, and all other tuples are missing. Figure 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 $\mathrm{R}(b, a)$ is undetermined). The truth of $\exists y\, R(a, y)$ and $\exists y\, \mathrm{R}(b, y)$ is justified by the existing tuples $\mathrm{R}(a, a)$ and $\mathrm{R}(b, a)$, respectively. Applying the translation $\mathrm{Tr}_{\mathrm{EXPL}\rightarrow\mathbb{N}[X,\bar{X}]}$, 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, $Q := \mathrm{TL}_{\varphi\rightarrow Q}(\varphi)$, and I_π the instance corresponding to π as defined above. Then*

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

Proof We prove the theorem by induction over the structure of the input formula φ for a given valuation ν of $\textsc{free}(\varphi)$. For the full proof, see [27]. □

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 $\textsc{Expl}(P, \psi, I)$ 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 $\psi_{n,s} = \textsc{Why}\, \mathtt{Q}(n, s)$ 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., 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 pushdown 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 [25]). For an input P, we use $P_{Unified}$ to denote the result of this unification.

8.2 Add annotations based on success/failure

For $\textsc{Why}\ Q(t)$ ($\textsc{Whynot}\ Q(t)$), 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 [25]). 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 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. 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. For EDB predicates, we create firing rules that have the firing version of the predicate in the head and the EDB predicate in the body.

Example 11 Consider the annotated program in Example 10 for the question $\psi_{n,s} = \text{WHY}\,\mathtt{Q}(n, s)$. 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

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

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

with firing rules that have inverted annotations (e.g., the goal $\neg\,\mathtt{T}(n, s)^T$ is replaced with $\mathtt{F}_{\mathtt{T},F}(n, s)$). Furthermore, we introduce firing rules for EDB tuples (three rules at the bottom).

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 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 $\mathtt{R.A}$ 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 12 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

$$
\begin{aligned}
\mathtt{F}_{\mathtt{Q},F}(s,n) &:- \neg\, \mathtt{F}_{\mathtt{Q},T}(s,n)\\
\mathtt{F}_{\mathtt{Q},T}(s,n) &:- \mathtt{F}_{\mathtt{r_1},T}(s,n,Z)\\
\mathtt{F}_{\mathtt{r_1},F}(s,n,Z,V_1,V_2,\neg V_3) &:- \mathtt{F}_{\mathtt{Q},F}(s,n), \mathtt{F}_{\mathtt{T},F/T}(s,Z,V_1),\\
&\qquad \mathtt{F}_{\mathtt{T},F/T}(Z,n,V_2), \mathtt{F}_{\mathtt{T},F/T}(s,n,V_3)\\
\mathtt{F}_{\mathtt{r_1},T}(s,n,Z) &:- \mathtt{F}_{\mathtt{T},T}(s,Z), \mathtt{F}_{\mathtt{T},T}(Z,n), \mathtt{F}_{\mathtt{T},F}(s,n)\\
\mathtt{F}_{\mathtt{T},F/T}(s,Z,true) &:- \mathtt{F}_{\mathtt{T},T}(s,Z)\\
\mathtt{F}_{\mathtt{T},F/T}(s,Z,false) &:- \mathtt{F}_{\mathtt{T},F}(s,Z)\\
\mathtt{F}_{\mathtt{T},T}(s,Z) &:- \mathtt{T}(s,Z)\\
\mathtt{F}_{\mathtt{T},F}(s,Z) &:- dom_{\mathtt{T.toCity}}(Z), \neg\, \mathtt{T}(s,Z)
\end{aligned}
$$

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

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 ith goal exists for this binding iff V_i is true. The failure status of the ith 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.

Algorithm 1 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 [25]. 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 do or do not exist. 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 existence 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.

Algorithm 1 Create Firing Rules

```
1:  procedure CREATEFIRINGRULES($P_{Annot}$, $\psi$)
2:    $P_{Fire} \leftarrow []$
3:    $state \leftarrow typeof(\psi)$
4:    $Q(t) \leftarrow$ PATTERN($\psi$)
5:    $todo \leftarrow [Q(t)^{state}]$
6:    $done \leftarrow \{\}$
7:    while $todo \neq []$ do                         ▷ create rules for a predicate
8:      $R(t)^\sigma \leftarrow$ POP($todo$)
9:      INSERT($done$, $R(t)^\sigma$)
10:     if ISEDB($R$) then
11:       CREATEEDBFIRINGRULE($P_{Fire}$, $R(t)^\sigma$)
12:     else
13:       CREATEIDBNEGRULE($P_{Fire}$, $R(t)^\sigma$)
14:       $rules \leftarrow$ GETRULES($R(t)^\sigma$)
15:       for all $r \in rules$ do                    ▷ create firing rule for $r$
16:         $args \leftarrow args(head(r))$
17:         $args \leftarrow args :: (args(body(r)) - args(head(r)))$
18:         CREATEIDBPOSRULE($P_{Fire}$, $R(t)^\sigma$, $r$, $args$)
19:         CREATEIDBFIRINGRULE($P_{Fire}$, $R(t)^\sigma$, $r$, $args$)
20:   return $P_{Fire}$
```

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 12.

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 (5th and 6th 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.

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 ith 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 structure of a program. For the proof, see [26] or [27]. □

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 13 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

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

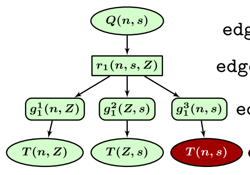
Fig. 10 Example firing rules with connectivity checks

$\mathtt{F}_{\mathtt{r}_1,T}(n, s, Z)$ 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 $\mathtt{T}$, 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 1st 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 3rd and 4th 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 kth goal in the body of r_i, we create a rule $\mathtt{FC}_{\mathtt{r_j},\mathtt{r}_\mathtt{i}^\mathtt{k},\mathtt{T}}(\vec{X})$ as follows. We unify the variables of the kth 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 $f_{\mathtt{T}}^{T}(n, s)$. Each rule firing corresponds to a fragment of $\mathcal{PG}(P, I)$. For example, one such fragment is shown in Fig. 11 (left). Such a substructure is created through a set of rules:



$$\texttt{edge}(f^{T}_{\texttt{Q}}(n, s), f^{T}_{\texttt{r}_1}(n, s, Z)) :- \texttt{F}_{\texttt{r}_1,T}(n, s, Z)$$
$$\texttt{edge}(f^{T}_{\texttt{r}_1}(n, s, Z), f^{T}_{\texttt{g}_1^1}(n, Z)) :- \texttt{F}_{\texttt{r}_1,T}(n, s, Z)$$
$$\texttt{edge}(f^{T}_{\texttt{g}_1^1}(n, Z), f^{T}_{\texttt{T}}(n, Z)) :- \texttt{F}_{\texttt{r}_1,T}(n, s, Z)$$
$$\texttt{edge}(f^{T}_{\texttt{g}_1^3}(n, s), f^{F}_{\texttt{T}}(n, s)) :- \texttt{F}_{\texttt{r}_1,T}(n, s, Z)$$

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)

- 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

Example 14 Consider the firing rules with connectivity joins from Example 13. 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 $\texttt{Q}(n, s)$ to a successful rule node $r_1(n, s, Z)$ is created by the topmost 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., $\texttt{F}_{\texttt{r}_1,F}(s, n, Z, V_1, V_2, \neg V_3)$).

8.6 $\mathcal{K}$-explanations

To compute one of the $\mathcal{K}$-explanations types introduced in Sect. 6.2, we only have to adapt the rules generating the edge relation. As an example, we present the modifications for computing $\text{EXPL}_{\mathsf{Which}(X)}$ (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 $\text{EXPL}_{\mathsf{Which}(X)}$, we create only one type of rule that connects tuple nodes for the head predicate to EDB tuple nodes. We use $\mathbb{GP}^{\mathsf{Which}(X)}_{P,\psi}$ to denote the program generated in this way for an input program P, and a PQ ψ.

Example 15 Consider the graph fragment for r_1 in Fig. 11 (left) without rule and goal nodes. The rule that creates the edge between $\texttt{Q}(n, s)$ and $\texttt{T}(n, Z)$ is

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

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

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* $\text{EXPL}(P, \psi, I)$.

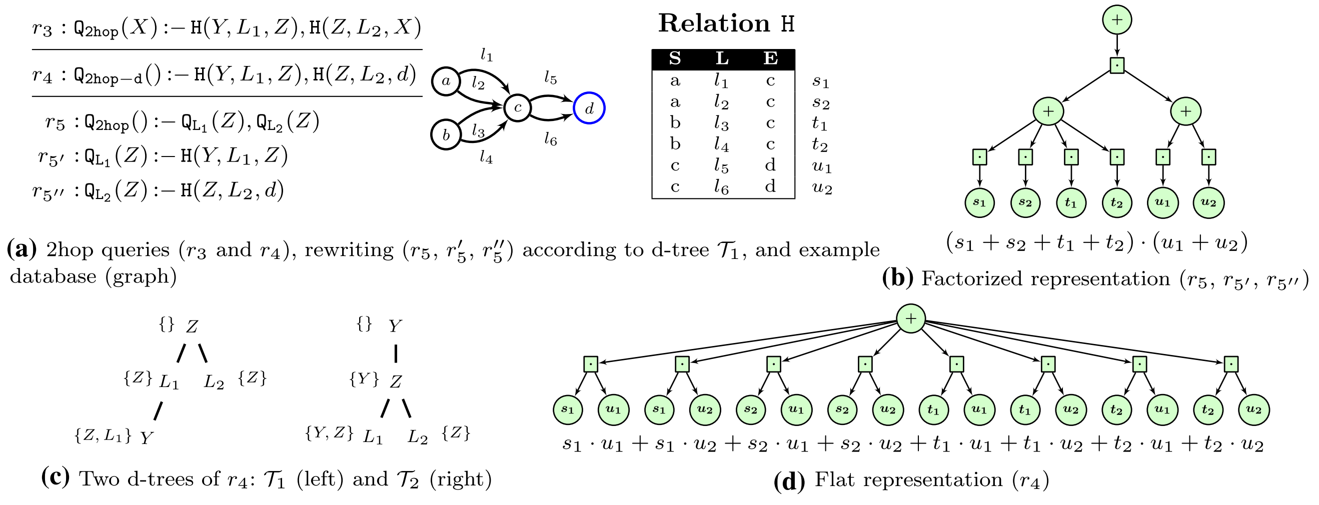
Proof To prove Theorem 5, we have to show that (1) only edges from $\mathcal{PG}(P, I)$ are in $\mathbb{GP}_{P,\psi}(I)$ and (2) the program returns precisely the set of edges of explanation $\text{EXPL}(P, \psi, I)$. The full proof is presented in [26] and in our accompanying report [27]. □

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* $\text{EXPL}_{\mathsf{Which}(X)}(P, \psi, I)$.

Proof We prove Theorem 6 by induction over the structure of a program as in the proof of Theorem 4. The full proof is presented in our technical report [27]. □

9 Factorization

For provenance polynomials, we can exploit the distributivity law of semirings to generate factorizations of provenance [33] 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-labeled graph in Fig. 12a. The provenance polynomial for the query result $\texttt{Q}_{\texttt{2hop}}(d)$ 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 "·" nodes for rule derivations and "+" nodes for IDB tuples. Figure 12b shows 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 [34] 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 [34], and how to choose 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


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

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

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

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

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

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

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

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

(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)

(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

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 Sect. 11).

Factorized representations In [33,34], 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 [33], we denote a singleton $\{(a)\}$ as a. Factorization can be applied to compactly represent relations and 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. [34] introduced *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 [33]. 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 labeled 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 [34] 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. Figure 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 [34], such representations are called *d-representations* (d-rep). Analogous to how f-trees define the structure of f-reps,

[4] In [34], relational algebra is used to express queries and nodes of f-trees represent equivalence classes of attributes which in Datalog correspond to query variables.

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 [34] is that, for a given d-tree $\mathcal{T}$ for a query Q, the size of d-rep of Q over a database 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 [34] 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, choosing the one with the lowest value of $s^{\uparrow}$.

Example 16 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 each result tuple with its provenance polynomial. 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 $\text{EXPL}_{\mathbb{N}[X]}$ graph for the user question 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 $\psi :=$ WHY $Q(t)$, to generate a Datalog query Q_{rewr} such that, for any database I, we have that $\text{EXPL}_{\mathbb{N}[X]}(Q_{rewr}, \psi, I)$ encodes $\mathbb{N}[X](Q_{rewr}, I, t)$ for each $t \in \text{MATCH}(\psi)$ factorized according to $\mathcal{T}$. We first unify the query with the PQ as described in Sect. 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 $\mathrm{Q_Y}(X \cup key(X))$ 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 $\mathrm{Q_1}(X_1,\ldots,X_n) :- \mathrm{Q_2}(X_1,\ldots,X_n)$ with the rule deriving $\mathrm{Q_2}$ (in our translation, there will be exactly one rule with head $\mathrm{Q_2}$).

Example 17 Consider the question WHY $\mathrm{Q_{2hop}}(d)$ 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:

$$r_{Q_{2hop}} : \mathrm{Q_{2hop}}() :- \mathrm{Q_Z}() \qquad r_Z : \mathrm{Q_Z}() :- \mathrm{Q_{L_1}}(Z), \mathrm{Q_{L_2}}(Z)$$
$$r_{L_1} : \mathrm{Q_{L_1}}(Z) :- \mathrm{Q_Y}(Z, L_1) \qquad r_Y : \mathrm{Q_Y}(Z, L_1) :- \mathrm{H}(Y, L_1, Z)$$
$$r_{L_2} : \mathrm{Q_{L_2}}(Z) :- \mathrm{H}(Z, L_2, d)$$

Applying the optimizations introduced above, we merge the rules $r_{Q_{2hop}}$ with r_Z (the head $\mathrm{Q_Z}$ 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 Sect. 8 to generate $\mathrm{EXPL}_{\mathbb{N}[X]}(Q_{rewr}, \psi, I)$ encoding the d-rep of $\mathbb{N}[X](Q_{rewr}, I, t)$ for each $t \in \mathrm{MATCH}(\psi)$.

Example 18 Continuing with Example 17, 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 topmost 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 $\mathrm{Q_{L_1}}(c)$ $(s_1 + s_2 + t_1 + t_2)$ and the right branch corresponds to the two derivations of $\mathrm{Q_{L_2}}(c)$ $(u_1 + u_2)$. 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 2 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

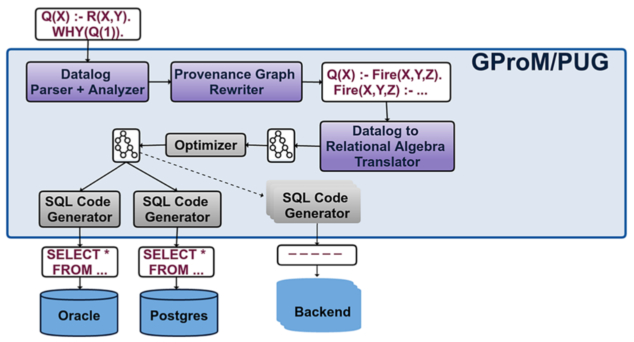

Q(X) :- R(X,Y). WHY(Q(1)).
GProM/PUG
Datalog Parser + Analyzer
Provenance Graph Rewriter
Q(X) :- Fire(X,Y,Z). Fire(X,Y,Z) :- ...
Datalog to Relational Algebra Translator
Optimizer
SQL Code Generator
SQL Code Generator
SQL Code Generator
SELECT * FROM ...
SELECT * FROM ...
Oracle
Postgres
Backend

Fig. 13 PUG implementation in GProM

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 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 Sect. 8 are applied to create the program $\mathbb{GP}_{P,\psi}$ for the input program P and the provenance question ψ which computes $\mathrm{EXPL}(P, \psi, I)$ (analogously, $\mathrm{EXPL}_{\mathcal{K}}(P, \psi, I)$ 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 [32] 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 [26] for details and an example.

11 Experiments

We evaluate the performance of our solution over a coauthor 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

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

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

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

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

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

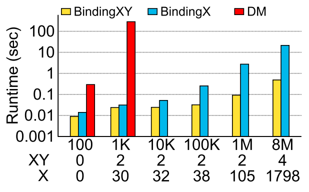
$$r_6 : \texttt{partNotAsia}(X) :- \texttt{PART}(A, X, B, C, D, E, F, G, H), \texttt{PARTSUPP}(A, I, J, K, L), \texttt{SUPPLIER}(I, M, N, O, P, Q, R), \texttt{NATION}(O, S, T, U), \neg\, \texttt{R}_1(T, \text{‘ASIA’})$$
$$r_{6'} : \texttt{R}_1(T, Z) :- \texttt{REGION}(T, Z, V)$$

$$r_7 : \texttt{suppCust}(N) :- \texttt{SUPPLIER}(A, B, C, N, D, E, F), \texttt{CUSTOMER}(G, H, I, N, J, K, L, M)$$

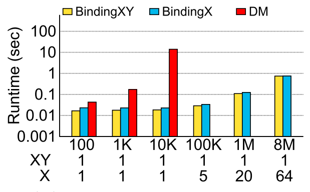
Fig. 14 DBLP and TPC-H queries for experiments

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



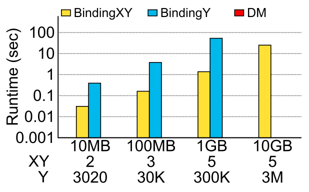
(a) Runtime of `only2hop`



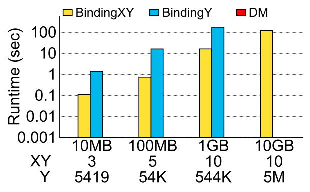
(b) Runtime of `XwithYnotZ`

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

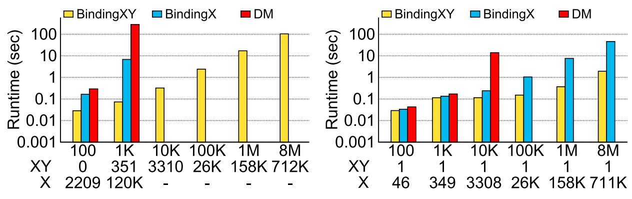
Fig. 16 Why questions: DBLP (top), TPC-H (bottom)

`Direct Method (DM)`, because it directly constructs the full provenance graph; (2) we compare our approach for Lineage ($\text{EXPL}_{\text{Which}(X)}$) 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 (Sect. 9). We have created subsets of the DBLP dataset with 100, 1 K, 10 K, 100 K, 1 M, and 8 M co-author pairs (tuples). For the TPC-H benchmark, we used database sizes 10 MB, 100 MB, 1 GB, and 10 GB. All experiments were run on a machine with 2 × 3.3 GHz 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 (DBMS X is the default). Unless stated otherwise, each experiment was repeated 100 times (we stopped executions that ran longer than 10 min) 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 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. Figure 16c, f shows which variables are bound by the provenance ques-

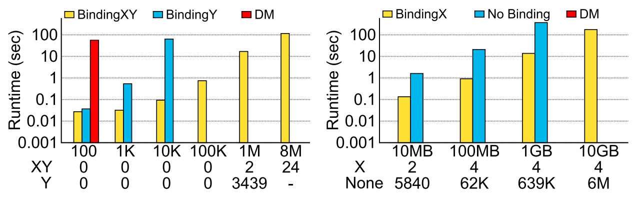


(a) Runtime of `only2hop` **(b)** Runtime of `XwithYnotZ`

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

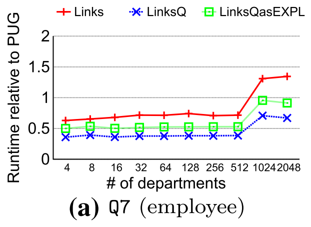
Fig. 18 Why questions for queries with negation

tions (PQs). Figure 16a, b shows 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, 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, e (for r_4 and r_5, respectively) shows 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, b, 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, b, 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, e, 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^{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^{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 $\text{EXPL}_{\text{Which}(X)}$). 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`



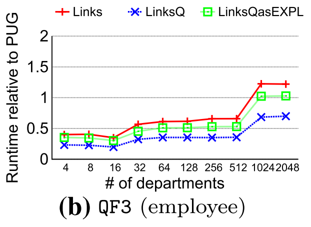
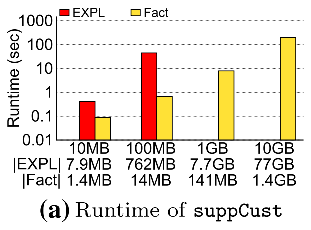


Fig. 19 Comparing Which(X) in PUG with Links



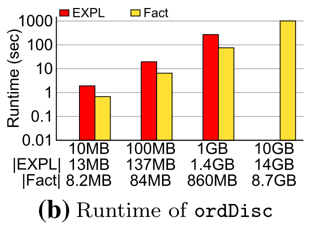


Fig. 20 Explanations versus factorized explanations

has three attributes (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, b, 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 (Sect. 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-min 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 [28]) to deal with the large size of explanations for missing answers. We plan to also support query-based explanations [2–4,40] and more expressive query languages (e.g., aggregation).

Acknowledgements This work was supported by NSF Awards OAC-{1640864, 1541450} and SMA-1637155. Opinions and findings expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

1. Arab, B., Gawlick, D., Radhakrishnan, V., Guo, H., Glavic, B.: A generic provenance middleware for database queries, updates, and transactions. In: TaPP (2014)
2. Bidoit, N., Herschel, M., Tzompanaki, K.: Immutably answering why-not questions for equivalent conjunctive queries. In: TaPP (2014)
3. Bidoit, N., Herschel, M., Tzompanaki, K., et al.: Query-based why-not provenance with NedExplain. In: EDBT, pp. 145–156 (2014)
4. Chapman, A., Jagadish, H.V.: Why not? In: SIGMOD, pp. 523–534 (2009)
5. Cheney, J., Chiticariu, L., Tan, W.: Provenance in databases: why, how, and where. Found. Trends Databases **1**(4), 379–474 (2009)
6. Damásio, C.V., Analyti, A., Antoniou, G.: Justifications for logic programming. In: Logic Programming and Nonmonotonic Reasoning, pp. 530–542 (2013)
7. Deutch, D., Gilad, A., Moskovitch, Y.: Selective provenance for datalog programs using top-k queries. PVLDB **8**(12), 1394–1405 (2015)
8. Deutch, D., Milo, T., Roy, S., Tannen, V.: Circuits for datalog provenance. In: ICDT, pp. 201–212 (2014)
9. Fehrenbach, S., Cheney, J.: Language-integrated provenance. Sci. Comput. Programm. **155**, 103–145 (2017)
10. Flum, J., Kubierschky, M., Ludäscher, B.: Total and partial well-founded datalog coincide. In: ICDT, pp. 113–124 (1997)
11. Glavic, B., Köhler, S., Riddle, S., Ludäscher, B.: Towards constraint-based explanations for answers and non-answers. In: TaPP (2015)
12. Glavic, B., Miller, R.J., Alonso, G.: Using sql for efficient generation and querying of provenance information. In: Tannen, V., Wong, L., Libkin, L., Fan, W., Tan, W.C., Fourman, M. (eds.) In Search of Elegance in the Theory and Practice of Computation, pp. 291–320. Springer, Berlin (2013)
13. Grädel, E., Tannen, V.: Semiring provenance for first-order model checking (2017). arXiv:1712.01980

14. Green, T.: Containment of conjunctive queries on annotated relations. Theory Comput. Syst. **49**(2), 429–459 (2011)
15. Green, T., Karvounarakis, G., Tannen, V.: Provenance semirings. In: PODS, pp. 31–40 (2007)
16. Green, T.J., Aref, M., Karvounarakis, G.: Logicblox, platform and language: a tutorial. In: Datalog in Academia and Industry, pp. 1–8. Springer, Berlin (2012)
17. Green, T.J., Karvounarakis, G., Ives, Z.G., Tannen, V.: Update exchange with mappings and provenance. In: VLDB, pp. 675–686 (2007)
18. Green, T.J., Tannen, V.: The semiring framework for database provenance. In: PODS, pp. 93–99 (2017)
19. Herschel, M., Diestelkämper, R., Lahmar, H.B.: A survey on provenance: What for? what form? what from? VLDB J **9**(3), 1–26 (2017)
20. Herschel, M., Hernandez, M.: Explaining missing answers to SPJUA queries. PVLDB **3**(1), 185–196 (2010)
21. Huang, J., Chen, T., Doan, A., Naughton, J.: On the provenance of non-answers to queries over extracted data. In: VLDB, pp. 736–747 (2008)
22. Karvounarakis, G., Green, T.J.: Semiring-annotated data: queries and provenance. SIGMOD Rec. **41**(3), 5–14 (2012)
23. Köhler, S., Ludäscher, B., Smaragdakis, Y.: Declarative datalog debugging for mere mortals. In: Datalog 2.0: Datalog in Academia and Industry, pp. 111–122 (2012)
24. Köhler, S., Ludäscher, B., Zinn, D.: First-order provenance games. In: Tannen, V., Wong, L., Libkin, L., Fan, W., Tan, W.C., Fourman, M. (eds.) Search of Elegance in the Theory and Practice of Computation, pp. 382–399. Springer, Berlin (2013)
25. Lee, S., Köhler, S., Ludäscher, B., Glavic, B.: Efficiently computing provenance graphs for queries with negation. Technical Report CoRR (2016). arXiv:1701.05699
26. Lee, S., Köhler, S., Ludäscher, B., Glavic, B.: A SQL-middleware unifying why and why-not provenance for first-order queries. In: ICDE, pp. 485–496 (2017)
27. Lee, S., Ludäscher, B., Glavic, B.: Pug: A framework and practical implementation for why and why-not provenance (extended version). Technical Report CoRR (2018). arXiv:1808.05752
28. Lee, S., Niu, X., Ludäscher, B., Glavic, B.: Integrating approximate summarization with provenance capture. In: TaPP (2017)
29. Meliou, A., Gatterbauer, W., Moore, K., Suciu, D.: The complexity of causality and responsibility for query answers and non-answers. PVLDB **4**(1), 34–45 (2010)
30. Meliou, A., Gatterbauer, W., Suciu, D.: Reverse data management. PVLDB **4**(12), 1490–1493 (2011)
31. Meliou, A., Suciu, D.: Tiresias: The database oracle for how-to queries. In: SIGMOD, pp. 337–348 (2012)
32. Niu, X., Kapoor, R., Glavic, B., Gawlick, D., Liu, Z.H., Krishnaswamy, V., Radhakrishnan, V.: Provenance-aware query optimization. In: ICDE, pp. 473–484 (2017)
33. Olteanu, D., Závodnỳ, J.: Factorised representations of query results: size bounds and readability. In: ICDT, pp. 285–298. ACM (2012)
34. Olteanu, D., Závodnỳ, J.: Size bounds for factorised representations of query results. ACM Trans. Database Syst. (TODS) **40**(1), 2 (2015)
35. Riddle, S., Köhler, S., Ludäscher, B.: Towards constraint provenance games. In: TaPP (2014)
36. Roy, S., Orr, L., Suciu, D.: Explaining query answers with explanation-ready databases. Proc. VLDB Endow. **9**(4), 348–359 (2015)
37. Roy, S., Suciu, D.: A formal approach to finding explanations for database queries. In: SIGMOD (2014)
38. Senellart, P.: Provenance and probabilities in relational databases. ACM SIGMOD Rec. **46**(4), 5–15 (2018)
39. Tannen, V.: Provenance analysis for FOL model checking. ACM SIGLOG News **4**(1), 24–36 (2017)
40. Tran, Q.T., Chan, C.-Y.: How to conquer why-not questions. In: SIGMOD, pp. 15–26 (2010)
41. Wu, E., Madden, S.: Scorpion: explaining away outliers in aggregate queries. PVLDB **6**(8), 553–564 (2013)
42. Wu, Y., Zhao, M., Haeberlen, A., Zhou, W., Loo, B.T.: Diagnosing missing events in distributed systems with negative provenance. In: SIGCOMM, pp. 383–394 (2014)
43. Xu, J., Zhang, W., Alawini, A., Tannen, V.: Provenance analysis for missing answers and integrity repairs. IEEE Data Eng. Bull. **41**(1), 39–50 (2018)
44. Zhou, W., Sherr, M., Tao, T., Li, X., Loo, B.T., Mao, Y.: Efficient querying and maintenance of network provenance at internet-scale. In: SIGMOD, pp. 615–626 (2010)