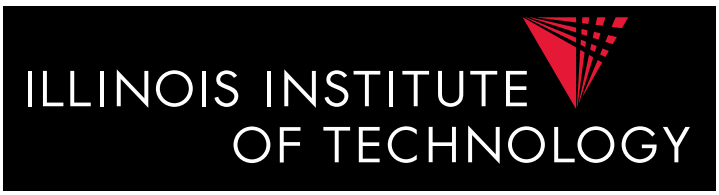


An Efficient Implementation of Game Provenance in DBMS

Seokki Lee, Yuchen Tang, Sven Köhler, Bertram Ludäscher, Boris Glavic

IIT DB Group Technical Report
IIT/CS-DB-2015-02

2015-10

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

An Efficient Implementation of Game Provenance in DBMS

Seokki Lee* Yuchen Tang* Sven Köhler† Bertram Ludäscher‡ Boris Glavic*
*Illinois Institute of Technology. {slee195@hawk.iit.edu, ytang34@hawk.iit.edu, bglavic@iit.edu}
‡University of Illinois at Urbana-Champaign. {ludaesch@illinois.edu}
†University of California at Davis. {svkoehler@ucdavis.edu}

***Abstract*—Explaining why a certain answer is in the result of a query or why it is missing from the result is important for many applications including auditing, debugging data, and answering hypothetical questions about data. Both types of questions, i.e., *why* provenance and *why-not* (missing answer) provenance have been studied extensively. *Provenance games*, a game-theoretic approach to provenance, can provide a unified view on why and why-not provenance. The power of provenance games, however, comes at the cost of complexity: a direct approach for computing game provenance instantiates a large game graph with all possible tuples that can be formed from values of the active domain using the rules of a query evaluation game. In this work, we present a new relational database approach for computing a much smaller subgraph of the provenance game that is *relevant* to the user's provenance question. In addition to avoiding the evaluation of a non-stratified *win-move* Datalog program, we also exclude parts of the game graph early on if we can determine that they will not be relevant to explain the user question. We present an implementation in the GProM provenance database middleware. Our experimental evaluation demonstrates that the approach scales to large instances and significantly outperforms the direct method.**

$r_0 : \mathtt{Q}(X, Y) :- \mathtt{Train}(X, Z), \mathtt{Train}(Z, Y)$

Train

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

City

city	state	country
new york	nystate	usa
chicago	illlinois	usa
washington dc	dc	usa
seattle	washington	usa

Fig. 1: Example database and query

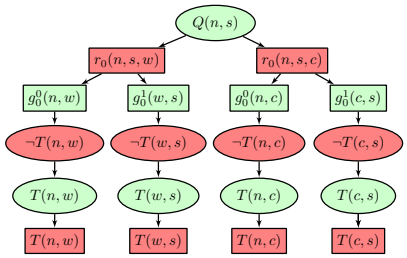


Fig. 2: Provenance game explaining why $\mathtt{Q}(n, s)$

I. Introduction

Provenance for relational queries records how results of a query depend on the query's inputs. This type of information can be used to explain *why* (and *how*) a result is derived by a query over a given database. Recently, approaches have been developed that use provenance-like techniques to answer the question why a tuple (or pattern describing potential tuples) is *missing* from the query result. However, the two problems of computing provenance and explaining missing answers have been treated in isolation. In this work, we argue that both problems are instances of the problem of computing provenance for queries with negation. Intuitively, asking why a tuple t is absent from Q is equivalent to asking why t is present in $\neg Q$. Thus, a provenance model that supports queries with negation should naturally support why-not provenance. We demonstrate that *provenance games*, a game-theoretical formalization of provenance for first-order queries (i.e., non-recursive Datalog with negation) satisfies these desiderata. Unfortunately, a direct implementation of provenance games is prohibitively expensive. In this work, we develop a new approach for solving provenance games for a user query in an efficient bottom-up fashion. We present the first practical implementation unifying why and why-not provenance using our GProM provenance middleware system that utilizes a relational database backend to execute first-order (non-recursive Datalog) queries. Furthermore, we prove that the provenance computed by our improved approach is correct, i.e., corresponds to the result obtained using the direct approach.

Example 1. *Consider the database in Fig. 1 storing information about cities (relation* `City`*) and train connections (relation* `Train`*) in the US. The Datalog rule r_0 computes which cities can be reached with one transfer. To clearly present the graph, abbreviations are used as follow: T = Train; n = New York; s = Seattle; w = Washington DC and c = Chicago. Given the result of this query, a user may be interested to know why they are able to reach Seattle from New York (why $\mathtt{Q}(n, s)$) with one transfer, or why it is not possible to reach Seattle from Chicago in the same fashion (why-not $\mathtt{Q}(c, s)$). The provenance games for these examples are shown in Fig. 2 and Fig. 3. Here, we only explain the intuition; see Section III for further details. Seattle can be reached from New York by either stopping in Washington DC or in Chicago. These two options correspond to two successful instantiations of rule r_0 with $X = n$, $Y = s$, and $Z = w$ (respectively, $Z = c$) (Fig. 2, 2nd line). A rule instantiation is successful if all grounded goals in the rule body evaluate to true. In the provenance game, this is represented through goal nodes that are connected to the rule node and to relation nodes for the goals' predicates (Fig. 2, 3rd line). Existing tuples are represented as relation nodes which are won (green) whereas missing tuples are represented as relation nodes which are lost (red) (Fig. 2, 4th line and 5th line). A rule node is lost if the*

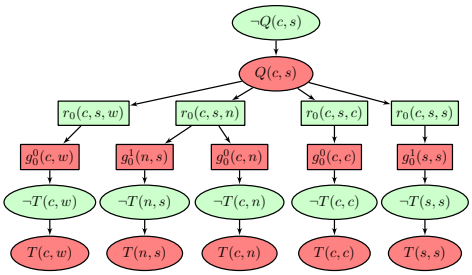


Fig. 3: Provenance game explaining why-not $\mathtt{Q}(c,s)$

body of the rule can not be refuted (i.e., the rule instantiation is successful) and won if it can (the instantiation failed). The game-theoretic foundation of this provenance model will be explained further in Section III. Finally, existing tuples are connected to EDB rule nodes (Fig. 2, 6th line). In summary, the example provenance game explains the existence of tuple $\mathtt{Q}(n,s)$ based on two successful instantiations of rule r_0 which are justified through existing `Train` *tuples in the EDB. The game provenance model supports first-order queries (i.e., Datalog with negation, but no recursion). As explained above, this means that the model can also explain missing answers. The example tuple $\mathtt{Q}(c,s)$ is missing from the query results, because all rule instantiations using values from the active domain $adom(I)$ of the database instance I that could have derived this tuple (with head $\mathtt{Q}(c,s)$) have failed. In the example, there are four failed instantiations of rule r_0 corresponding to four cities that can not be used as intermediate stops on the way from Chicago to Seattle. A rule is lost if at least one goal in the body is lost. Thus, in the provenance game, only failed goals are connected to the failed rule binding explaining a missing answer. For instance, we cannot reach Seattle from Chicago with an intermediate stop in Washington DC (Fig. 3 the first failed rule from the left in 3rd line), because there is no direct connection from Chicago to Washington DC whereas the direct connection between Washington DC and Seattle exists. Failed positive goals in the body of a failed rule are explained by missing tuples (lost relation nodes). In this example, these are relation nodes that are not connected to EDB rule nodes.*

A. Problem Definition

The problem we address in this work is how to explain the existence or absence of (sets of) tuples using provenance games. The two questions presented in the previous example used constants only, but the user can also ask questions involving variables, e.g., for a question $Q(n,X)$ we would return all explanations for existing or missing tuples where the first attribute is n, i.e., why or why-not can city X be reached from New York with one transfer.

Definition 1 (Explanation)**.** *Let P be a first-order query expressed as a Datalog program, Q be an IDB predicate, and I an instance. A why/why-not question is an atom $Q(t)$ where $t=(v_1,\ldots,v_n)$ is a tuple consisting of variables and constants from the active domain of I. We say an atom $Q(t')$* matches *an atom $Q(t)$, if we can unify $Q(t')$ with $Q(t)$ by replacing variables in t with variables or constants from t'. The* ***explanation*** $\textsc{Expl}(P,Q(t),I)$ *for $Q(t)$ according to P and I, is the subgraph of the provenance game for P and I containing only nodes that are connected to at least one node where $Q(t')$ matches $Q(t)$. We use $\textsc{Why}(Q(t))$ and $\textsc{Whynot}(Q(t))$ to denote a question where the user is only interested in existing (won) or respective missing (lost) tuples that match $Q(t)$.*

B. Overview and Contributions

The existing approach for computing provenance games [21] allows for uniform treatment of provenance and missing answer problems and provides a meaningful semantics for negation. The approach requires instantiation of the whole game for the instance, evaluation of a non-stratified Datalog program using well-founded semantics [11] over this graph, removal of edges that correspond to bad moves in the game, i.e., moving to a position won by the other player, and, finally, determining subgraphs related to the user question. While this approach is correct, it is prohibitively expensive. A critical problem is the size of the instantiated game which is $\mathcal{O}(\|adom(I)\|^k)$ where k is the maximal number of variables of rules in the input program and $adom(I)$ is the active domain of the database instance I. As an example consider a database with 100 values and a rule with 5 variables. There are in the order of $100^5 = 10^{10}$ nodes in the instantiated game. Typically, most of the nodes in the instantiated game will not end up being in the explanation for the user questions, because they are not connected in the final provenance game graph. In our approach, we employ several optimizations that reduce the amount of disconnected nodes that are created. First, our solution exploits information encoded in the user question, e.g., based on the constants in t, only certain nodes can be connected to $Q(t)$ in the provenance game. For instance, in the running example, $\mathtt{Q}(n,s)$ can only be connected to instantiations of rule r_0 with $X=n$ and $Y=s$. In turn, this implies that we are only interested in train tuples $\mathtt{Train}(n,Z)$ and $\mathtt{Train}(Z,s)$. Our approach propagates this type of information in a top-down traversal of the program. Additionally, based on the user question, we may be able to predetermine that we are only interested in won or lost states for certain nodes types (e.g., in the example, we are only interested in existing `Train` relation tuples). In addition, by simultaneously instantiating and solving the game, we avoid the execution of a non-stratified program and only create edges in the graph that are relevant for the user questions.

In summary, the main contributions of this work are:

- We demonstrate that the recently introduced provenance games model can serve as a unified framework for provenance of queries with negation and why-not provenance, and discuss the relationship of this model to provenance and missing answer approaches.
- We present an efficient approach for computing the explanation (provenance games) for a why or why-not question using Datalog. In contrast to the solution introduced in [21], our approach exploits information encoded in the question and avoids unnecessary work by simultaneously instantiating and solving the game.
- We formally prove the correctness of our algorithm for computing the provenance game for a user question.
- We present a full implementation of our approach in the GProM provenance database middleware that

compiles provenance game computations expressed in Datalog into relational algebra, optimizes these relational algebra expressions, and translates the optimized expression into SQL code that can be executed by a standard relational database backend.

The remainder of the paper is organized as follows. In Section II, we discuss related work. We introduce provenance games in Section III and demonstrate how to compute provenance games efficiently in Section IV. We present our implementation in Section V, discuss experimental results in Section VI, and conclude in Section VII.

II. Related Work

There are several lines of work related to the research presented in this paper. Obviously, game provenance has strong connections to other provenance models for relational queries, most importantly the semiring annotation framework, and approaches for explaining missing answers.

Database Provenance. There is a large body of work on models for provenance of database queries (e.g., see [7], [19]). The semiring annotation framework has developed into a quasi-standard for representing the provenance of positive relational algebra (and, thus, positive non-recursive Datalog queries). In this model, tuples in a relation are annotated with elements from a commutative semiring $\mathcal{K}$. It has been shown that by choosing certain semirings the K-relational model extends several extensions of the relational databases (including set-semantics, bag-semantics, and event-tables). More importantly for our purpose, this model also subsumes multiple relational provenance models. An essential property of the K-relational model is that there exists a semiring, the semiring of polynomials over a set of variables X with natural number coefficients, which generalizes all other semirings. This semiring $\mathbb{N}[X]$ is often called the provenance polynomials semiring. It has been shown in [21] that provenance games for positive queries generalize provenance semirings, and, thus all other provenance models expressible as semirings. They are closely related to boolean circuit representations of semiring provenance for Datalog programs [9] - both models explicitly share common subexpressions in the provenance. Exploring the relationship of provenance games for queries with negation and m-semirings (semirings with support for set difference) is an interesting avenue for future work. Justifications for outputs of logic programs [22] are also closely related. Recently, an algebraic model which uses semirings but with explicit negative tokens [8] which supports Datalog programs with negation has been studied.

Why-not and Missing Answers. Approaches for explaining missing answers, i.e., explaining why expected tuples are not in the results of a query can be classified based on whether they explain a missing answer by the query [3], [4], [3], [24], [6] (i.e., which operators did filter out tuples that would have contributed to the missing answer) or by the input data [17], [18] (i.e., what tuples need to be inserted into the database to turn the missing answer into an answer). The missing answer problem was first stated for query-based explanations in the seminal paper by Chapman et al. [6]. Huang et al. [18] first introduced an instance-based approach. Since then, several techniques have been developed to exclude spurious explanations, to support larger classes of queries [17], to combine instance and query-based explanations [16], and for distributed systems based on Datalog in Y! [25]. The approaches for instance-based explanations (with the exception of Y!) have in common that they treat the missing answer problem as a view update problem: the missing answer is a tuple that should be inserted into a view corresponding to the query and this insert has to be translated into the database instance. An explanation is then one particular solution to this view update problem. Many approaches do not return a random answer, but instead try to optimize for minimal side-effects on the base tables or views (requiring view maintenance). In contrast to these approaches, provenance games explain a missing answer by enumerating all failed rule derivations that would have caused the answer to be in the result. Y! applies a resolution approach to explains missing answers top-down, but also limits the result to one explanation. Solutions for the missing answer problem as defined in previous work can be extracted from the provenance game for an missing answer. So in a sense, provenance games generalize these approaches (for the class of queries supported by provenance games, e.g., we do not currently support aggregation). Interestingly, recent work has shown that it may be possible to generate more concise summaries of provenance games [12], [23] which is particularly useful for negation and missing answers to deal with the potentially large size of the resulting provenance games. This approaches are complementary to our work and in fact would benefit from it, because these approaches would benefit from our efficient computation of provenance games.

Computing Provenance Declaratively. The concept of rewriting a Datalog program using firing rules to capture provenance as variable bindings of rule instantiations was used for provenance-based debugging of positive Datalog queries [20] and later adopted for using provenance to inject faults to discovering bugs in distributed systems specified in Daedalus [1] (a distributed Datalog language). These rules are also equivalent to Datalog implementations of the semiring model in Orchestra [15], LogicBlox [14], and the Perm system's [13] rewrites for SPJ queries. An extension for distributed systems using either full propagation or reference based provenance has been proposed for ExSPAN [26]. Extensions of this type of rules for negation are one of the main enablers of our approach for computing provenance games.

III. Provenance Games

Provenance Games as introduced by Köhler et al. [21] provide explanations $\textsc{Expl}(P, Q(t), I)$ for all atoms $Q(t)$ in non-recursive Datalog$^\neg$ programs P. They express evaluation as a 2-player game in a game-theoretic sense. The provenance game consists of 1) a *game graph* and 2) a set of simple rules. In the game graph, nodes are positions in the game while directed edges represent potential moves that the player in the originating node can choose from. The game rules are as follows: a game starts in any position and players take turns by choosing an outgoing edge to follow until the game ends in a node without an outgoing edge. At this point, the player who has to make the next move has *lost*. Since each player is assumed to play perfectly, a node is *won* (the player won) if following any outgoing edge reaches a lost node. A node is *lost* if all outgoing edges lead to won nodes (there is no way for this player to win). As shown in [21], games for non-recursive

Datalog$^\neg$ programs (FO queries) have no cycles and thus do not allow for draws. Now, we illustrate how to construct and solve a provenace game graph according to Köhler et al. [21].

A. Evaluation Games

Evaluation game is constructed from a program and a given EDB instance using the following definition.

Definition 2 (Evaluation Game)**.** *The evaluation game $G_{Q,I}$ for a program Q and EDB instance I is a game graph defined over the grounded program. The nodes of $G_{Q,I}$ are:*

- *A postive and a negative relation node $R(x)$ and $\neg R(x)$ for each ground atom $R(x)$*
- *A rule node $r_i(x,y)$ and corresponding goal nodes $g_i^j(x_j)$ or $g_i^j(x_j,y_j)$ for each grounded rule r_i. Here, x_j and y_j denotes the arguments of the j^{th} goal in the grounded rule.*
- *An EDB rule node $r_R(x)$ for every tuple $R(x) \in I$.*

The edges of $G_{Q,I}$ are: $(\neg R(x), R(x))$ for each relation node pair $R(x)$ and $\neg R(x)$; $(R(x), r_i(x,y))$ for each relation node $R(x)$ and rule node $r_i(x,y)$ where the grounded head of grounded rule $r_i(x,y)$ is $R(x)$; $(r_i(x,y), g_i^j(x_j))$ for each rule node $r_i(x,y)$ and corresponding goal node $g_i^j(x_j)$; $(g_i^j(x_j), \neg R(x))$ for each node for a negative goal g_i^j where R is the goal's predicate; $(g_i^j(x_j), R(x))$ for each node for a positive goal g_i^j where R is the goal's predicate; $(R(x), r_R(x))$ for each relation node $R(x)$ and rule node $r_R(x)$ if $R(x) \in I$.

Fig. 4 shows the instantiated game for query $Q_{\texttt{ABC}}$ with input database $I = \{B(a,b), B(b,a), C(a)\}$:

$$r_1: \quad \texttt{A}(X) :- \underbrace{\texttt{B}(X,Y)}_{g_1}, \underbrace{\neg\,\texttt{C}(Y)}_{g_2} \qquad (Q_{\texttt{ABC}})$$

B. Solved Evaluation Games

An evaluation game graph, modelled as an edge relation $\texttt{move}(X,Y)$, can be solved by evaluating the non-stratified Datalog program $\texttt{win}(X) :- \texttt{move}(X,Y), \neg\,\texttt{win}(Y)$ under the well-founded semantics [11]. In the *solved game*, every node is marked as either won or lost. Formally, given an evaluation game $G_{Q,I}(V,E)$, we use $G^{\Gamma}_{Q,I}(V,E,\Gamma)$ to denote the solved game where $\Gamma : V \to \{won, lost\}$ is a function that maps nodes to their won/lost state. Any relation node that is marked as won corresponds to an existing tuple in the IDB or EDB and a lost relation node corresponds to a missing tuple. Thus, solving the instantiated game for a program Q and instance I is akin to evaluating Q over I. The solved game for the running example in this section is also shown in Fig. 4. Here, we see that I has a winning strategy for e.g., $A(a)$, $B(b,a)$, and $C(a)$, as these are part of instance.

C. Deleting Bad Moves aka Provenance Games

A solved evaluation game already provides all information about the provenance of program Q. However, parts of the game graph are irrelevant for explaining existing and missing tuples in the IDB relations. Intuitively, if we look at the subgraph rooted at an (negated) IDB relation node, this subgraph

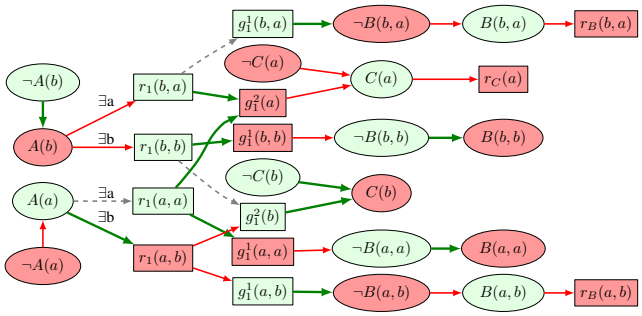


Fig. 4: Solved instantiated game $G^{\lambda}_{Q_{\texttt{ABC}},I}$. Lost positions are (dark) red; won positions are (light) green. Provenance edges (good moves) are solid; bad moves are dashed. $A(a)$ (resp. $A(b)$) is true (resp. false), indicated by position value W (resp. L). The game provenance $\Gamma(A(a))$ explains why $A(a)$ is true.

explains how the tuple was produced (or failed to be produced). Some edges in such a subgraph correspond to non-optimal moves while we require players to play optimal. We use an edge-labeling function $\lambda : V \times V \to \{good, bad\}$ to distinguish different types of moves. $\lambda(x_1,x_2) = bad$ if both x_1 and x_2 are won, otherwise $\lambda(x_1,x_2) = good$.

The *provenance game* is derived from a solved game by excluding bad moves. The game provenance for a particular tuple (relation node) is defined as the node's transitive closure, i.e., for a question $\texttt{Q}(t)$, the subgraph rooted at the nodes matching question (in case t only has constants, this will be exactly one node). For WHY (WHYNOT) questions, only subgraphs rooted at won (lost) nodes are considered.

Definition 3 (Provenance Game)**.** *Let $G^{\lambda} = (V, M, \gamma)$ be a solved game. The* provenance game Γ *is the good-labeled subgraph of G^{λ}. For $x \in V$, we define $\Gamma(x)$ as the subgraph of Γ, reachable via λ edges. Note that for a user question $\texttt{Q}(t)$, $\Gamma(\texttt{Q}(t))$ is the explanation for $\texttt{Q}(t)$ according to Definition 1.*

D. Naive Provenance Game Computation

Having presented some basic background on provenance games, we are now ready to explain the top-down approach for computing a provenance game that was introduced in [21]. A Datalog program is generated that computes the move graph of the instantiated game according to Definition 2. Fig. 5 shows the result for the example program $Q_{\texttt{ABC}}$. Here, domain $\texttt{d}$ denotes a relation storing $adom(I)$, and skolem functions are used to create node identifiers.

The game instantiated in this fashion is then solved by evaluating it together with the win-move program using well-founded semantics. The provenance game is obtained by discarding bad edges, $\lambda(X,Y) = bad$, and computing a restricted move relation M' using the following Datalog rule:

$$\texttt{M}'(X,Y) :- \texttt{M}(X,Y), \neg\,\texttt{win}(X)$$
$$\texttt{M}'(X,Y) :- \texttt{M}(X,Y), \neg\,\texttt{win}(Y)$$

These rules ensure that at least one node of each edge is lost. To obtain the provenance for a question, the graph needs to be restricted to nodes reachable from any node matching the

Atoms A,B, **and** C

$\mathtt{M}(f_{\neg \mathtt{A}}(X), f_{\mathtt{A}}(X)) \quad :- \ \mathtt{d}(X).$
$\mathtt{M}(f_{\neg \mathtt{B}}(X,Y), f_{\mathtt{B}}(X,Y)) \quad :- \ \mathtt{d}(X), \mathtt{d}(Y).$
$\mathtt{M}(f_{\neg \mathtt{C}}(X), f_{\mathtt{C}}(X)) \quad :- \ \mathtt{d}(X).$

IDB A **via rule** r_1

$\mathtt{M}(f_{\mathtt{A}}(X), f_{\mathtt{r}_1}(X,Y)) \quad :- \ \mathtt{d}(X), \mathtt{d}(Y).$
$\mathtt{M}(f_{\mathtt{r}_1}(X,Y), f_{\mathtt{g}_1^1}(X,Y)) \quad :- \ \mathtt{d}(X), \mathtt{d}(Y).$
$\mathtt{M}(f_{\mathtt{r}_1}(X,Y), f_{\mathtt{g}_1^2}(Y)) \quad :- \ \mathtt{d}(X), \mathtt{d}(Y).$
$\mathtt{M}(f_{\mathtt{g}_1^1}(X,Y), f_{\neg \mathtt{B}}(X,Y)) \quad :- \ \mathtt{d}(X), \mathtt{d}(Y).$
$\mathtt{M}(f_{\mathtt{g}_1^2}(X), f_{\mathtt{C}}(X)) \quad :- \ \mathtt{d}(X).$

EDB B **and** C

$\mathtt{M}(f_{\mathtt{B}}(X,Y), f_{\mathtt{r}_{\mathtt{B}}}(X,Y)) \quad :- \ \mathtt{B}(X,Y).$
$\mathtt{M}(f_{\mathtt{C}}(X), f_{\mathtt{r}_{\mathtt{C}}}(X)) \quad :- \ \mathtt{C}(X).$

Fig. 5: Datalog Rules for Computing a Relational Encoding of an Instantiated Evaluation Game for $Q_{\mathtt{ABC}}$.

question. A Datalog program is generated that computes a reduced move relation $\mathtt{M}_\mathtt{u}$ for the why question $\mathtt{Q}(t)$:

$$\mathtt{M}_\mathtt{u}(f_\mathtt{Q}(t), Y) :- \mathtt{M}'(f_\mathtt{Q}(t), Y), \mathtt{Q}(t)$$
$$\mathtt{M}_\mathtt{u}(Y, Z) :- \mathtt{M}_\mathtt{u}(X, Y), \mathtt{M}'(Y, Z)$$

Note that in [21], the last two steps (computing M' and M_u) are not described in an algorithmically. We have chosen a straightforward implementation.

IV. Computing Provenance Games with Datalog

In Section III, we reviewed an elegant and simple solution for computing the relational encoding of a provenance game. While this solution is easy to understand, it is computationally expensive. In this section, we present a novel approach to compute provenance games for a given query Q efficiently. We also show how this approach can be implemented using a standard relational database. In particular, we generate a datalog program that computes the game bottom-up where pieces of the game are instantiated and solved simultaneously. In our solution, we exploit the fact that only certain nodes can be connected to atoms covered by the user questions. For example, consider a program $\mathtt{Q}(X) :- \mathtt{R}(X,Y)$ and a user question $\mathtt{Q}(1)$. Since the user asks about $X = 1$, only those parts of the graph which agree with $X = 1$ are instantiated, solved, and shown in the provenance game (PG) graph.

Given a question $Q(t)$ for an instance I and program P, our approach constructs a Datalog program $\mathbb{GP}(Q(t), P)$ which evaluated over I will return the edge relation of $\Gamma(Q(t))$, the subgraph of the provenance game for P and I restricted to nodes connected to $Q(t)$. We can then construct a graphical representation of provenance (e.g., the graphs shown in this paper were produced by our implementation). The program $\mathbb{GP}(Q(t), P)$ is generated using the following steps:

1. We first unify the program with the user question $\mathtt{Q}(t)$ by propagating the constants in t throughout the program in a top-down fashion. Replacing variables in the program with constants in this way is correct, because only nodes with these bindings may be connected to $Q(t)$ in the PG.

2. Afterwards, we determine for which nodes in the graph we can predetermine whether they will be won or lost based on whether the user question atom is won or lost. We model this information as adornments on atoms in the rules of the program and propagate these adornments top-down.

3. Based on the annotated and unified game created in the previous steps, we generate rules that capture the variable bindings for successful and failed rule instantiations. The adornments enable us to determine whether we can focus on successful or failed instantiations only. The results of these *firing rules* are sufficient for creating the PG graph.

4. Being an answer to a firing rule is a necessary, but not sufficient condition for the PG graph fragment corresponding to this rule binding to be connected to the user question. To guarantee that only relevant fragments are returned, we have to filter out rule bindings by checking connectivity.

5. Finally, we create rules that generate the edge relation of the PG based on the rule bindings information.

In the following, we will explain each step in detail and illustrate its application based on the following example.

Example 2. *Consider the following query over the train relation* ($\mathtt{T}$) *from the running example computing which cities can be reached from each other through exactly two hops in the train connection graph, but not directly. Assume the user asks why* $\mathtt{only2hop}(n,s)$*, i.e., why can Seattle be reached from New York in this fashion.*

$$r_1 : \mathtt{only2hop}(X,Y) :- \mathtt{T}(X,Z), \mathtt{T}(Z,Y), \neg \mathtt{T}(X,Y)$$

A. Unify the Program with the User Question

Intuitively, the node $\mathtt{only2hop}(n,s)$ in the PG graph is only connected to rule instantiations which return $\mathtt{only2hop}(n,s)$. For instance, if variable X is bound to c in an instantiation of rule r_1 then this rule cannot return (n,s). This reasoning can be applied recursively to replace variables in Datalog rules with constants. That is why we unify the rules in the program top-down with the user question.

Example 3. *Given the user question why* $\mathtt{only2hop}(n,s)$*, we unify the single rule* r_1 *using the assignment* $(X = n, Y = s)$*:*

$$r_1^{(X=n,Y=s)} : \mathtt{only2hop}(n,s) :- \mathtt{T}(n,Z), \mathtt{T}(Z,s), \neg \mathtt{T}(n,s)$$

For EDB relations we introduce a new rule for each binding with head $\mathtt{U}_\mathtt{R}$ *and the unified atom as the body.*

$$T^{(X=n)} : \quad \mathtt{U}_\mathtt{T}(n,Z) :- \mathtt{T}(n,Z)$$
$$T^{(Y=s)} : \quad \mathtt{U}_\mathtt{T}(Z,s) :- \mathtt{T}(Z,s)$$
$$T^{(X=n,Y=s)} : \quad \mathtt{U}_\mathtt{T}(n,s) :- \neg \mathtt{T}(n,s)$$

As the example illustrated above, we may have to create multiple partially unified versions of a rule or EDB atom. Note that the results of this step are not meant to be evaluated. We simply use this unified program to provide information about potential bindings that may lead to PG fragments connected to the user question. The pseudocode for this step is shown in Algorithm 1. For all rules creating the user question predicate, we unify the head of each rule with the user question. This may cause some of the variables occurring in the body to be replaced by constants. The body atoms of the unified rules are then appended to a todo list. The algorithm processes each

Algorithm 1 Unify Program With User Question

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

atom on the todo list by applying the same type of unification to each rule creating this atom until the todo list is empty. We keep track of which unified head atoms have been processed so far to avoid creating the same unified rules more than once (the same unified atom may appear as a goal in several rules).

B. Add Won/Lost Adornments

The user question provides us with the information that an atom is won or lost (is absent or present in the IDB). Based on this fact, we can infer restrictions on the won/lost state of nodes that are connected to the user question atom in the PG graph. [1] We store this information as adornments T, F, and F/T on atoms in rules. Here T indicates that we are only interested in won nodes, F that we are only interested in lost nodes, and F/T that we are interested in both.

Example 4. *Since we know that* $\texttt{only2hop}(n,s)$ *is won, we can determine the won/lost states of rules in the unified program. Adornments are determined based on a top-down propagation seeded with the user question. For instance, in the example the won status of* $\texttt{Q}(n,s)$ *implies that only won goal nodes can be connected to the relation node. Note that this adornment does not imply that this rule* r_1 *would be won for every* Z*, it only indicates that it is sufficient to focus on won rule instantiations since lost ones can not be connected to the user question.*

$$r_1^{(X=n,Y=s),T}: \quad \texttt{only2hop}(n,s)^T :- \texttt{T}(n,Z)^T, \texttt{T}(Z,s)^T, \neg\, \texttt{T}(n,s)^T$$

We now propagate the adornments of the goals in r_1 *throughout the program. Note that for negated goals we propagate the opposite adornment. For instance, for* $\neg\, \texttt{T}(n,s)^T$ *we annotated the head of rule* $T^{(X=n,Y=s)}$ *with* F*.*

$$T^{(X=n),T}: \quad \texttt{U}_\texttt{T}(n,Z)^T :- \texttt{T}(n,Z)$$
$$T^{(Y=s),T}: \quad \texttt{U}_\texttt{T}(Z,s)^T :- \texttt{T}(Z,s)$$
$$T^{(X=n,Y=s),F}: \quad \texttt{U}_\texttt{T}(n,s)^F :- \neg\, \texttt{T}(n,s)$$

Atoms may occur in both negative and positive goals of the rules of program. If an atom may be both won or lost, then we create two adorned versions of the rules for this atom. This is denoted by using F/T adornments for the EDB rules corresponding to these relations. The use of these adornments will become more clear in the next subsection when we introduce firing rules, i.e., rules that capture the variable bindings of failed and successful rule instantiations.

Algorithm 2 is used to compute these adornments. Similar to the unification step, we start by adorning each rule with the user question predicate in the head. A rule is adorned by first adorning the head and then propagating the head adornment to the body goals. As in step 1, we keep a queue `todo` of atoms to process and record which atoms have been processed already (set `done`). For predicates in the body of a rule adorned with T, all positive goals are appended to the todo list with T adornments. For negated goals, F adornments are used. The rationale behind this approach is that if we are only interested in existing tuples derived by a rule r, then this implies that we are only interested in existing tuples for the predicates of positive goals and missing tuples for the predicate of negated goals. For rules with F adornments, we are only interested in missing head predicate tuples. As explained above, we have to compute both existing and missing tuples for goals to capture all different possibilities of failures (represented as F/T adornments). Thus, for predicates in the body of a rule adorned with F, all goal predicates are pushed to the end of the todo list with F/T adornments. For each goal, if this goal's predicate has not been processed before and the goal predicate is IDB (EDB predicates do not occur as head predicate of a rule and, thus, do not need to be adorned), it is appended to the todo list.

After all rules have been adorned, we remove rules adorned with F or T if there exists the same rule with adornment F/T, because for rules adorned with F/T we will compute both successful and failed bindings as explained in the next subsection. The function SwitchState, used in the algorithm, switches F and T and maps F/T onto itself.

C. Creating Firing Rules

To be able to compute the relevant part of the PG graph for a user question, we need to determine successful and failed rule instantiations. Each rule instantiation paired with the information whether it is successful given the database instance (and which goals are failed in case it is not successful) corresponds to a certain PG subgraph. Successful rule instantiations are always part of the PG graph whereas failed rule instantiations only appear in the PG graph if their head atom is lost, i.e., there does not exist any successful rule instantiation of any rule with this head atom. To capture the variable bindings of successful rule instantiations, we create "firing rules". A "firing rule" consists of the body of the original rule in the program and a new head predicate that contains all variables used in the rule. For instance, the firing rule for rule $r: \texttt{Q}(X) :- \texttt{R}(X,Y)$ would be $\texttt{F}_{\texttt{r},\texttt{T}}(X,Y) :- \texttt{R}(X,Y)$.

Example 5. *Consider the adorned program for question* $\texttt{only2Hop}(n,s)$*. We generate the firing rules shown in Fig. 6. The firing rule for* $r_1^{(X=n,Y=s),T}$ *is derived from* r_1 *by adding* Z*, the only existential variable in* r_1*, to the head, renaming the head predicate as* $\texttt{F}_{\texttt{r}_1,\texttt{T}}$*, and replacing each goal with its firing version. Note that negated goals are replaced with firing rules that have inverted adornments. For example, the goal* $\neg\, \texttt{T}(n,s)$ *is replaced with* $\texttt{F}_{\texttt{T},\texttt{F}}(n,s)$*.*

[1] If we do not trust the user, we can run a simple query to determine whether the user atom is won or lost.

Algorithm 2 Compute Won/Lost Adornments

```
1: procedure AdornProgram(P_U, Q(t))
2:     state ← typeof(Q(t))
3:     todo ← [Q(t)^state]
4:     done ← {}
5:     P_A = []
6:     while todo ≠ [] do
7:         a ← pop(todo)
8:         state ← typeof(a)
9:         insert(done, a)
10:        rules ← getUnRulesForAtom(P, a)
11:        for all r ∈ rules do
12:            adRule ← adornRule(r, state)
13:            P_A ← P_A :: adRule
14:            for all g ∈ body(adRule) do
15:                if state = F then
16:                    newstate ← F/T
17:                else
18:                    newstate ← state
19:                if isNegated(g) then
20:                    newstate ← switchState(state)
21:                if g^newstate ∉ done ∧ isIDB(g) then
22:                    todo ← todo :: g^newstate
23:    for all r ∈ P_A do
24:        if typeof(r) = F/T then
25:            P_A ← removeAdoredRules(P_A, r, {F, T})
26:    return P_A
```

$$\begin{aligned}
\mathtt{F}_{\mathtt{r}_1,\mathtt{T}}(n,s,Z) &:- \mathtt{F}_{\mathtt{T},\mathtt{T}}(n,Z), \mathtt{F}_{\mathtt{T},\mathtt{T}}(Z,s), \mathtt{F}_{\mathtt{T},\mathtt{F}}(n,s)\\
\mathtt{F}_{\mathtt{T},\mathtt{T}}(n,Z) &:- \mathtt{T}(n,Z)\\
\mathtt{F}_{\mathtt{T},\mathtt{T}}(Z,s) &:- \mathtt{T}(Z,s)\\
\mathtt{F}_{\mathtt{T},\mathtt{F}}(n,s) &:- \neg\, \mathtt{T}(n,s)
\end{aligned}$$

Fig. 6: Example Firing Rules

$$\begin{aligned}
\mathtt{F}_{\mathtt{Q},\mathtt{F}}(1) &:- \neg\, \mathtt{F}_{\mathtt{Q},\mathtt{T}}(1)\\
\mathtt{F}_{\mathtt{Q},\mathtt{T}}(1) &:- \mathtt{F}_{\mathtt{r},\mathtt{T}}(1,Y,Z)\\
\mathtt{F}_{\mathtt{r},\mathtt{F}}(1,Y,Z,V_1,V_2) &:- \mathtt{F}_{\mathtt{Q},\mathtt{F}}(1), \mathtt{F}_{\mathtt{R},\mathtt{F}/\mathtt{T}}(1,Z,V_1), \mathtt{F}_{\mathtt{S},\mathtt{F}/\mathtt{T}}(Z,Y,V_2)\\
\mathtt{F}_{\mathtt{r},\mathtt{T}}(1,Y,Z) &:- \mathtt{F}_{\mathtt{R},\mathtt{F}/\mathtt{T}}(1,Z,true), \mathtt{F}_{\mathtt{S},\mathtt{F}/\mathtt{T}}(Z,Y,true)\\
\mathtt{F}_{\mathtt{R},\mathtt{F}/\mathtt{T}}(1,Z,true) &:- \mathtt{F}_{\mathtt{R},\mathtt{T}}(1,Z)\\
\mathtt{F}_{\mathtt{R},\mathtt{F}/\mathtt{T}}(1,Z,false) &:- \mathtt{F}_{\mathtt{R},\mathtt{F}}(1,Z)\\
\mathtt{F}_{\mathtt{R},\mathtt{T}}(1,Z) &:- \mathtt{R}(1,Z)\\
\mathtt{F}_{\mathtt{R},\mathtt{F}}(1,Z) &:- adom(Z), \neg\, \mathtt{R}(1,Z)\\
\mathtt{F}_{\mathtt{S},\mathtt{F}/\mathtt{T}}(X_1,X_2,true) &:- \mathtt{F}_{\mathtt{S},\mathtt{T}}(X_1,X_2)\\
\mathtt{F}_{\mathtt{S},\mathtt{F}/\mathtt{T}}(X_1,X_2,false) &:- \mathtt{F}_{\mathtt{S},\mathtt{F}}(X_1,X_2)\\
\mathtt{F}_{\mathtt{S},\mathtt{T}}(X_1,X_2) &:- \mathtt{S}(X_1,X_2)\\
\mathtt{F}_{\mathtt{S},\mathtt{F}}(X_1,X_2) &:- adom(X_1), adom(X_2), \neg\, \mathtt{S}(X_1,X_2)
\end{aligned}$$

Fig. 7: Example Firing Rules for Why-not Question

For a failed rule instantiation where the head atom is a missing tuple, we need to know which goals are successful, because only failed goals are connected to the node representing this rule instantiation in the PG graph. To capture this information we add additional variables - V_i for goal g^i - to a firing rule's head that record which goal is succesful (corresponding to won respective lost goal node in the PG).

Algorithm 3 Create Firing Rules

```
1: procedure CreateFiringRules(P_A, Q(t))
2:     P_F ← []
3:     state ← typeof(Q(t))
4:     todo ← [Q(t)^state]
5:     done ← {}
6:     while todo ≠ [] do                ▷ create rules for a predicate
7:         R(t')^σ ← pop(todo)
8:         insert(done, R(t')^σ)
9:         if isEDB(R) then
10:            CreateEDBFiringRule(P_F, R(t)^σ)
11:        else
12:            CreateIDBNegPredRule(P_F, R(t)^σ)
13:            rules ← getRules(R(t')^σ)
14:            for all r ∈ rules do          ▷ create firing rule for r
15:                args ← (vars(r) − vars(head(r)))
16:                args ← args(head(r)) :: args
17:                CreateIDBPosPre-
   dRule(P_F, R(t')^σ, r, args)
18:                CreateIDBRuleFir-
   ingRule(P_F, R(t')^σ, r, args)
19:    return P_F
```

The body of a firing rule for failed rule instantiations is created by adding the negated head atom to the body (to check that the instantiation is part of an explanation for the failure to derive this atom and not one of the failed rule instantiations for a head atom that is part of the IDB) and by replacing all predicates in the body with their firing counterpart.

Example 6. *Consider a why-not question* $\mathtt{Q}(1)$ *for a program* $r : \mathtt{Q}(X) :- \mathtt{R}(X,Z), \mathtt{S}(Z,Y)$. *The firing rules are shown in Fig. 7 with the exception of the rules for* S *which are analog to the rules for* R. *Since* $\mathtt{Q}(1)$ *is lost, we are only interested in failed rule instantiations of* r *with* $X = 1$. *Furthermore, each rule node in the PG corresponding to such a rule instantiation will only be connected to lost subgoals. Thus, as mentioned above, we need to capture which goals are won/lost for each such rule instantiation. This is modelled through boolean variables* V_1 *and* V_2 *that are true if the corresponding goal is won and false otherwise. Thus, we also need firing rules for the relations mentioned in the subgoals that record both existing and absent tuples and use a boolean variable to record whether a tuple is present respective absent. IDB relation* $\mathtt{F}_{\mathtt{r},\mathtt{F}}(1,Y,Z,V_1,V_2)$ *will contain all variable bindings for instantiations of rule* r *such that* $Q(1)$ *is the head, the rule instantiation is failed, and the* i^{th} *goal is won/lost for this binding iff* V_i *is true/false. To produce all these bindings, we need rules capturing won and lost relation nodes for each subgoals for rule* r. *We denote such rules using an* T/F *adornment. For EDB relations, we will create two rules: one for the won and one for the lost case. For the lost case we use a predicate* $adom$ *to bind variables to all values in the activate domain (for safe negation).*

The algorithm for creating the firing rules for an adorned input program is shown as Algorithm 3. The algorithm maintains a list of adorned atoms that need to be processed that is initialized with the user question. For each such atom $R(t')^\sigma$ (here σ is the adornment of the atom) it creates firing rules for each rule r that has this atom as a head, for existing tuples for R, and a firing rule $\mathtt{F}_{\mathtt{R},\mathtt{F}}(t')$ for missing tuples (only if the

Algorithm 4 Create Firing Rules Subprocedures

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

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

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

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

atom is adorned with F/T or F).

EDB atoms. For an EDB atom $R(t)^T$ we use procedure CREATEEDBFIRERULE to create one rule $\mathtt{F}_{\mathtt{R},\mathtt{T}}(t) :- R(t)$ that "copies" the EBD relation R. 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 in the active domain and do not exist in R. This is achieved by adding goals $adom(X_i)$ to bind each such variable X_i to all values from $adom(I)$ and checking that the resulting tuple does not exist in R ($\neg R(t)$). If

$$\mathtt{F}_{\mathtt{r_1},\mathtt{T}}(n, s, Z) :- \mathtt{F}_{\mathtt{T},\mathtt{T}}(n, Z), \mathtt{F}_{\mathtt{T},\mathtt{T}}(Z, s), \mathtt{F}_{\mathtt{T},\mathtt{F}}(n, s)$$
$$\mathtt{FC}_{\mathtt{T},\mathtt{r}^1_1,\mathtt{T}}(n, Z) :- \mathtt{T}(n, Z), \mathtt{F}_{\mathtt{r_1},\mathtt{T}}(n, s, Z)$$
$$\mathtt{FC}_{\mathtt{T},\mathtt{r}^2_1,\mathtt{T}}(Z, s) :- \mathtt{T}(Z, s), \mathtt{F}_{\mathtt{r_1},\mathtt{T}}(n, s, Z)$$
$$\mathtt{FC}_{\mathtt{T},\mathtt{r}^3_1,\mathtt{F}}(n, s) :- \neg\, \mathtt{T}(n, s), \mathtt{F}_{\mathtt{r_1},\mathtt{T}}(n, s, Z)$$

Fig. 8: Example Firing Rules with Connectivity Checks

Algorithm 5 Add Connectivity Joins

procedure ADDCONNECTIVITYRULES(P_F, $Q(t)$)
  $P_{FC} \leftarrow []$
  $paths \leftarrow$ PATHSTARTINGIN($P_F, Q(t)$)
  for all $p \in paths$ **do**
    $p \leftarrow$ FILTERRULENODES(p)
    for all $e = (r_1(\vec{X_1})^{\sigma_1}, r_2(\vec{X_2})^{\sigma_2}) \in p$ **do**
      $goals \leftarrow$ GETMATCHINGGOALS(e)
      for all $g_i \in goals$ **do**
        $g_{new} \leftarrow$ UNIFYHEAD($\mathtt{F}_{\mathtt{r_1},\sigma_1}(t_1), g_i, \mathtt{F}_{\mathtt{r_2},\sigma_2}(t_2)$)
        $r_{new} \leftarrow \mathtt{FC}_{\mathtt{r_2},\mathtt{r}^i_1,\sigma_2}(t) :- body(\mathtt{F}_{\mathtt{r_2},\sigma_2}(t_2)), g_{new}$
        $P_{FC} \leftarrow P_{FC} :: r_{new}$
  return P_{FC}

the atom is adorned F/T, then two additional rules are added - one capturing existing and one missing tuples. A constant argument $true$ respective $false$ in the rules head is used to distinguish between existing and missing tuples.

IDB atoms. Firing rules for IDB predicates create an additional level of indirection by splitting each rule r^T : $R(t) :- g_1(\vec{X_1}), \ldots, g_n(\vec{X_n})$ into two rules: one representing rule bindings for r (with $\mathtt{F}_{\mathtt{r},\sigma}$ as head predicate and the firing versions of the goals as body) and one with the firing version of R as a head and $\mathtt{F}_{\mathtt{r},\sigma}$ as the body. Procedure CREATEIDB-NEGPREDRULE creates rules for tuples missing from an IDB predicate. For an IDB atom $R(t)^F$, i.e., tuples missing from R, we create a rule $\mathtt{F}_{\mathtt{R},\mathtt{F}}(t) :- \neg\, adom(X_1), \ldots, \mathtt{F}_{\mathtt{R},\mathtt{T}}(t)$. That is, a tuple is missing from R if it is not in R. We create one rule with head $\mathtt{F}_{\mathtt{R},\mathtt{T}}(t)$ for each rule with R as a head. The creation of these rules is described below. Atoms adorned with F/T are handled as described above for EDB atoms.

Rules. For the positive case, we create one rule $\mathtt{F}_{\mathtt{R},\mathtt{T}}(t') :- \mathtt{F}_{\mathtt{r},\mathtt{T}}(\vec{X})$ where $\vec{X}$ is the concatenation of t' with all existential variables from the body and a second rule with head $\mathtt{F}_{\mathtt{r},\mathtt{T}}(\vec{X})$ and the same body as r except that each goal is replaced with its firing version with appropriate adornment (e.g., T for positive goals). For rules adorned with F or F/T we create one additional rule:

$$\mathtt{F}_{\mathtt{r},\mathtt{F}}(\vec{X}, V'_1, \ldots, V'_n) :- \mathtt{F}_{\mathtt{R},\mathtt{F}}(t), \mathtt{F}_{\mathtt{pred(g_1)},\mathtt{F}/\mathtt{T}}(\vec{X_1}, V_1), \ldots$$

This rule captures instantiations of rules r for missing tuples. Here $V'_i = V_i$ for positive goals and $V'_i = \neg V_i$ for negated goals. Note that $\neg V$ denotes negating a boolean value in this case. The negated goal $\mathtt{F}_{\mathtt{R},\mathtt{F}}(t)$ is necessary, because even for existing tuples some rule instantiations may fail.

D. Connectivity Joins

As mentioned before, if a binding is returned by a firing rule, this is a necessary but not sufficient condition for the corresponding rule node to be connected to the user question

$$\texttt{edge}(f^T_Q(n,s), f^F_{r_1}(n,s,Z)) :- \texttt{F}_{\texttt{r}_1,\texttt{T}}(n,s,Z) \quad (Q \to r_1)$$
$$\texttt{edge}(f^F_{r_1}(n,s,Z), f^T_{g^1_1}(n,Z)) :- \texttt{F}_{\texttt{r}_1,\texttt{T}}(n,s,Z) \quad (r_1 \to g^1_1)$$
$$\texttt{edge}(f^F_{r_1}(n,s,Z), f^T_{g^2_1}(Z,s)) :- \texttt{F}_{\texttt{r}_1,\texttt{T}}(n,s,Z) \quad (r_1 \to g^2_1)$$
$$\texttt{edge}(f^F_{r_1}(n,s,Z), f^T_{g^3_1}(n,s)) :- \texttt{F}_{\texttt{r}_1,\texttt{T}}(n,s,Z) \quad (r_1 \to g^3_1)$$
$$\texttt{edge}(f^T_{g^1_1}(n,Z), f^F_{\neg T}(n,Z)) :- \texttt{F}_{\texttt{r}_1,\texttt{T}}(n,s,Z) \quad (g^1_1 \to \neg T)$$
$$\texttt{edge}(f^F_{\neg T}(n,Z), f^T_{T}(n,Z)) :- \texttt{F}_{\texttt{r}_1,\texttt{T}}(n,s,Z) \quad (\neg T \to T)$$
$$\texttt{edge}(f^T_{g^2_1}(Z,s), f^F_{\neg T}(Z,s)) :- \texttt{F}_{\texttt{r}_1,\texttt{T}}(n,s,Z) \quad (g^2_1 \to \neg T)$$
$$\texttt{edge}(f^F_{\neg T}(Z,s), f^T_{T}(Z,s)) :- \texttt{F}_{\texttt{r}_1,\texttt{T}}(n,s,Z) \quad (\neg T \to T)$$
$$\texttt{edge}(f^T_{g^3_1}(n,Z), f^F_{T}(n,s)) :- \texttt{F}_{\texttt{r}_1,\texttt{T}}(n,s,Z) \quad (g^3_1 \to T)$$
$$\texttt{edge}(f^T_{T}(n,Z), f^F_{r_T}(n,Z)) :- \texttt{FC}_{\texttt{T},\texttt{r}^1_1,\texttt{T}}(n,Z) \quad (T \to r_T)$$
$$\texttt{edge}(f^T_{T}(Z,s), f^F_{r_T}(Z,s)) :- \texttt{FC}_{\texttt{T},\texttt{r}^2_1,\texttt{T}}(Z,s) \quad (T \to r_T)$$

Fig. 9: Rules Creating the Edge Relation of EXPL($\texttt{Q}(n,s)$)

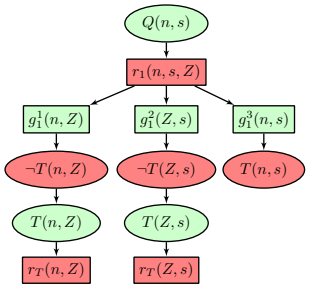


Fig. 10: Graph Structure for Explanation for $\texttt{Q}(n,s)$

in the PG graph. Thus, to guarantee that only nodes connected to the user question node are returned by our rules, we have to check whether they are actually connected.

Example 7. *Consider the firing rules for the why question* $\texttt{only2Hop}(n,s)$ *from previous examples. The corresponding rules with connectivity are shown in Fig. 8. All rule nodes corresponding to IDB tuples* $\texttt{F}_{\texttt{r}_1,\texttt{T}}(n,s,Z)$ *are guaranteed to be connected to the user question node. However, this does not hold for the nodes corresponding to IDB tuples returned by, e.g., rule* $T^{(X=n),T}$*, because a relation node* $\texttt{T}(n,c)$ *tuple for a constant* c *is only connected to the user question node iff it is part of a successful binding of rule* r_1*. That is there has to exist another hop tuple* (n,s) *but the hop relation does not contain a tuple* (n,s)*. To that end, we check connectivity from the user question node one hop at a time. In this example, we unify the head of each* $\texttt{T}$ *rule with the corresponding goal in the firing rule for rule* r_1*. For instance, for rule* $T^{(Y=s),T}$ *we unify with goal* $\texttt{F}_{\texttt{T},\texttt{T}}(Z,s)$ *to get*

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

The head of the unified rule is then added to the body of rule $T^{(Y=s),T}$*. We use* $\texttt{FC}_{\texttt{R},\texttt{r}^i,\texttt{T}}(\vec{X})$ *to denote the firing rule for* R *connected to the* i^{th} *goal of rule* r*. Note that for rules with only constants this connectivity check is unnecessary.*

The general algorithm (Algorithm 5) finds all paths in the game template for the input program starting in the node corresponding to the user question. For each edge (r,R) on such a path where r is a rule and R is an IDB or EDB relation node, we connect the rules corresponding the relation node $V(R)$ to this edge with the rules for rule r. Note that we compute these paths using the game template for the unified adorned program (i.e., corresponding to the result of the second step). Thus, as a first step, all other nodes are removed from a path p. After this step, every edge on a path connects a rule node to another rule node (where one of the rules may be an EDB rule node). For each combination of rules r_1 and r_2 on the current path, we determine which goals of r_1 match the head of r_2. For each such goal g, we unify the firing rule for r_1's variables for this goal with the variables of the firing rule of r_2 corresponding to the head of original rule r_2. For instance, in Fig. 8 the firing rule $\texttt{F}_{\texttt{T},\texttt{T}}(n,Z)$ matches the first goal of rule r_1. The unified head of the firing rule for r_1 is then added to the body of r_1 to create a connectivity join rule $\texttt{FC}_{\texttt{r}_2,\texttt{r}^i_1,\sigma_2}$ (assuming that the goal was the i^{th} goal). Effectively, these rules check connectivity of the rule nodes in the PG with the user question atom one hop at a time.

E. Computing the Edge Relation

The program created so far captures all information needed to generate the edge relation of the GP for an user question. To compute the edge relation, we use skolem functions to create node identifiers. The identifier of a node captures the type of the node (relation, rule, or goal), assignments from variables to constants, and the won/lost status of the node. For example, a relation node $\texttt{T}(n,s)$ that is won would be represented as $f^T_T(n,s)$. Each rule firing corresponds to a pattern in the PG graph as shown in Fig. 10 for the running example. Such a substructure is created through a set of rules:

- One rule creating edges between relation nodes for the head predicate and rule nodes
- One rule for each goal connecting a rule node to a node for that goal (for lost rules only the lost goals are connected)
- One rule for each goal connecting the goal node to the (negated) relation node for the goal's predicate (negated for positive goals only)
- One rule for each goal connecting the negated relation node for the goal's predicate to the positive relation node (for positive goals only)

For EDB firing rules, we only create one type of edge:

- One rule creating edges from a won relation node to won EDB node

The pseudocode for the Algorithm to create datalog rules that will return the edge relation is shown in Appendix B.

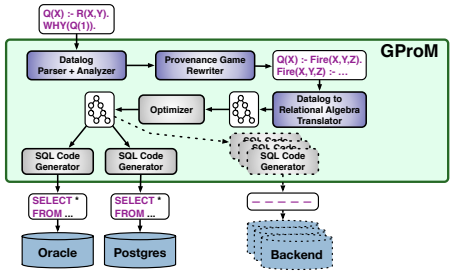


Fig. 11: Provenance Game Implementation in GProM

Example 8. *Consider the firing rules with connectivity joins from the previous example. Some of the rules for creating the edge relation for the explanation sought by the user are shown in Fig. 9. For example, the edges connecting the relation node* $\mathtt{Q}(n, s)$ *(the user question node) to a won rule node* $r_1(n, s, Z)$ *are created by the top-most rule. In the figure, the type of edges corresponding to a rule are shown to the right, e.g., the third rule creates edges between a rule node* $r_1(n, s, Z)$ *and its first goal* $g_1^1(n, Z)$.

F. Correctness and Complexity

We now analyze the complexity of our approach and state its correctness, i.e., the graph returned by the approach is the explanation for the user question.

Theorem 1 (Correctness)**.** *The graph returned by program* $\mathbb{GP}(Q(t), P)$ *over instance* I *is* $\textsc{Expl}(Q(t), P, I)$.

Proof: The proof is shown in Appendix A ■

The worst-case complexity of our approach is actually quite discouraging. In the worst case, the size of the program $\mathbb{GP}(Q(t), P)$ can be exponential in the size of the input program. This blowup may happen only in the first step (unification with user question) while the remaining steps only increate the program size by a constant factor.

Lemma 1 (Size of the Unified Program)**.** *Let* $Q(t)$ *be a user question over program* P *and instance* I*. The size of the program* P_U *produced by step 1 of our algorithm may be exponentially larger than* P.

Proof: The proof is shown in Appendix A ■

Note that we can avoid this blowup by simply removing the unification step. However, the unification step can provide useful selection conditions for why questions over positive programs and can reduce the exponential factor when computing why-not questions. In practise, we have yet to see any program where this step results in a super-linear increase in program size. Thus, for now we will assume that the increase in program size caused by the unification step is bounded by a constant factor. If this should ever become a concern, we could implement a safety check that stops unification if the increase in program size exceeds a threshold. Since we translate the provenance game construction into a relational algebra query with is at most constant factor larger than the input query, the standard complexity results for relational algebra queries apply (PSPACE combined complexity and PTIME data complexity). However, this result is not particularly interesting, because the naive method also exhibits the same bounds, since the well-founded evaluation of the win-move rule has a bounded number of iteration, i.e., we could express it as a non-recursive datalog program that is only a constant factor (the number of iterations) larger. The more interesting question is how the exponent in polynomial time runtime of the input program compares to the naive solution and our approach. For positive queries, the runtime of our approach is limited by the size of the instance and program whereas the naive approach runs in $\mathcal{O}(\|adom(I)\|^n)$ where n is the maximal $\|vars(r)\|$ for rules $r \in P$. For queries involving negation and why-not questions the naive approach exhibits the same runtime for why questions over positive queries. The size of an explanation for a why-not or query involving negation is bound by $\mathcal{O}(\|adom(I)\|^m)$ where m is the number of existential variables in F adorned rules. For provenance games involving negation the runtime of our approach is bound by the maximum of $\mathcal{O}(\|adom(I)\|^m)$ and the regular complexity of the remainder of the program. Since, $m \leq n$ and in practice often $m < n$, our approach is expected to outperform the naive approach in most cases even if we do not take additional factors such as the restriction to constants based on unification and connectivity filtering into account.

Proposition 1 (Complexity)**.** *Assume that the unification step of our approach only increase the program by a constant factor. If the program* P *runs in* $\mathcal{O}(\|I\|^n)$*, then the runtime of* $\mathbb{GP}(Q(t), P)$ *in bound by the maximum of* $\mathcal{O}(\|I\|^{2n})$ *and* $\mathcal{O}(\|adom(I)\|)^m$ *where* m *is the maximal number of existential variables in rules adorned with* F.

V. Implementation

As illustrated in Fig. 11, we implemented the aforementioned algorithm in a provenance middleware called GProM [2] that executes provenance requests using a relational database backend. The system was originally developed to support provenance requests for SQL (new components added to support game provenance are shown in blue). To support game provenance requests, we developed a parser for Datalog enriched with syntax for stating user provenance questions. Currently, the user can ask WHY and WHYNOT questions providing both the datalog program and the question as an input to the system. As a first step, the input program is parsed and semantically analyzed. Schema information is gathered by querying the catalog of the backend database (e.g., to determine whether an EDB predicate with the expected arity exists). Modules for accessing schema information are already part of the GProM system, but a new semantic analysis component had to be developed to support Datalog programs. If the input includes a user question $Q(t)$, then the algorithm presented in Section IV is applied to create a program $\mathbb{GP}(P, Q(t))$ which computes the edge relation of the provenance graph $\textsc{Expl}(P, Q(t), I)$.

This program is translated into a relational algebra graph (GProM uses algebra graphs instead of trees to allow for sharing of common sub-expressions). This algebra graph is then translated into SQL and send to the backend database for execution. This query returns the edge relation of the provenance game and then the system renders a graph representation (e.g., the examples shown in the introduction are actual results produced by the system). While it would certainly be possible to directly translate the Datalog program into SQL without the

$$\textsc{Tra}(Q) = \delta(\bigcup_{i \in \{j_1,\ldots,j_m\}} \textsc{Tra}(r_i)) \quad \text{(T1)}$$

$$\textsc{Tra}(r_i) = \Pi_{\vec{X}_i}(\textsc{Tra}(pos_{r_i}) \bowtie \textsc{Tra}(\hat{g}_i^1) \bowtie \ldots \bowtie \textsc{Tra}(\hat{g}_i^{\hat{n}_i})) \quad \text{(T2)}$$

$$\textsc{Tra}(pos_{r_i}) = \textsc{Tra}(g_i^1) \bowtie \ldots \bowtie \textsc{Tra}(g_i^{n_i}) \quad \text{(T3)}$$

$$\textsc{Tra}(R) = R \quad \text{(T4)}$$

$$\textsc{Tra}(g_i^j(\vec{X}_{i,j})) = \begin{cases} \rho_{\vec{X}_{i,j}}(\textsc{Tra}(R_{i,j})) & \text{if } g_i^j \text{ is a positive subgoal} \\ \Pi_{\vec{X}_{i,j}}(\textsc{Tra}(pos_{r_i})) - \textsc{Tra}(R_{i,j}) & \text{else} \end{cases} \quad \text{(T5)}$$

Fig. 12: Rules for Datalog $\rightarrow$ Relational Algebra Translation

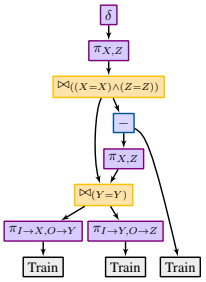


Fig. 13: `only2Hop` Translation

intermediate translation into relational algebra, we choose to introduce this step to be able to leverage the existing heuristic and cost-based optimizations for relational algebra graphs built into GProM and its library of algebra to SQL translators.

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

Let $P = \{r_1, \ldots, r_n\}$ be a non-recursive program where

$$r_i : R_i(\vec{X}_i) :- g_i^1(\vec{X}_{i,1}), \ldots, g_i^{n_i}(\vec{X}_{i,n_i}), \neg\, \hat{g}_i^1(\vec{X}'_{i,1}), \ldots, \neg\, \hat{g}_i^{\hat{n}_i}(\vec{X}'_{i,\hat{n}_i}))$$

(i.e., WLOG we assume negated goals are following the positive goals and negated goals are denoted as $\hat{g}$) and Q be the distinguished answer relation of P. Here g_i^j is a positive or negative subgoal. We use $R_{i,j}$ to denote the relation corresponding to g_i^j. Let Q be an IDB relation and R denote an EDB relation. Fig. 12 shows the definition of a function $\textsc{Tra}()$ that realizes the translation. Note that we assume that the translation caches results, i.e., a sub-expression is only translated once and the resulting algebra expression will be reused when needed. We now explain how to translate the individual parts of a program in detail.

Translating Relations. Access to an EDB relation is represented by a relation access in relational algebra (rule `T4`). We create a rule for each IDB relation Q as a union of the translations of all rules with this relation as a head (rule `T1`). Here r_{j_1} to r_{j_m} denote the rules in P with head predicate Q.

Translating Rules. A Datalog rule is translated as a join of the goals in the body followed by a projection on the head variables. Variables that are reused among goals become join conditions. For instance, the rule $\texttt{Q}(X, Z) :- \texttt{R}(X, Y), \texttt{S}(Y, Z)$ over `R` with attributes A_1 and A_2 and relation `S` with attributes A_3 and A_4 can be expressed in relational algebra as:

$$\Pi_{A_1,A_4}(R \bowtie_{A_2=A3} S)$$

We use the rename operator ρ which enables natural joins to translate a subgoal. For instance, a goal $\texttt{R}(X, Y)$ is translated as $\rho_{X,Y}R$. Also the joins of subgoals are processed in two steps: 1) all positive goals are joined (rule `T3`) and 2) then the result of this process is joined with the translation of all negative goals (rule `T2`). The reason for introducing this intermediate step is that the join of the positive goals will be used in the translation of a negated goal.

Translating Goals. Rule `T5` translates goals. A positive goal is translated as a renaming as explained above. A negated subgoal $\neg R(\vec{X})$ can be represented in relational algebra as $adom(\vec{X}) - R$ where $adom(\vec{X})$ is the active domain of the variables $\vec{X}$ restricted by the bindings in the rule. For example, for

$$\texttt{Q}(A, B) :- \texttt{R}(A, B), \neg\texttt{T}(B).$$

the active domain of B is restricted to the values occurring in the 2^{nd} attribute of relation R. Since Datalog rules are safe (every variable occurring in a negated goal will be bound through its occurrences in positive goals), the negated goal only needs to be tested for variable bindings produced by the positive part of the rule. We use this fact in the translation to express the negated goal as a set difference between bindings for the goals variables produced by the positive part pos_r of the rule r and the translation of the goal.

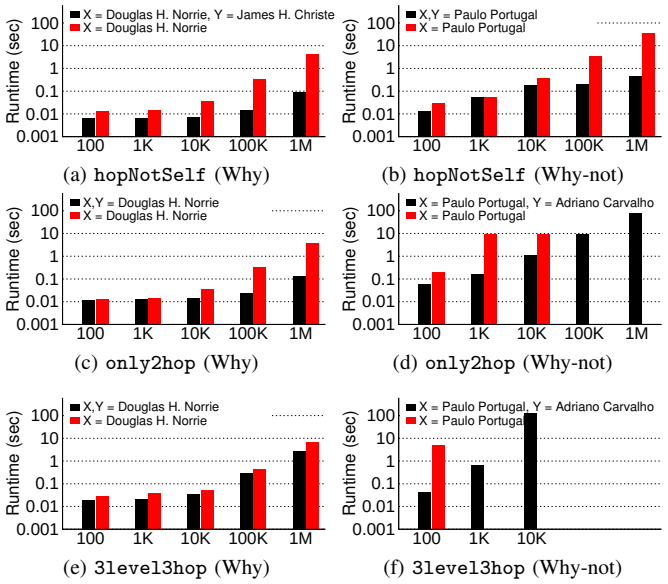
Example Translation. Consider the translation of `only2hop`.

$$r : \texttt{only2hop}(X, Y) :- \texttt{T}(X, Z), \texttt{T}(Z, Y), \neg\texttt{T}(X, Y)$$

The relational algebra graph of this rule is shown in Fig. 13. The translation of the first two goals are joined to form the variable bindings for the positive part of the query. The negated goal is translated as a set difference between the positive part projected on X, Z and the `Train` relation. The last three operators join the positive with the negative part, project on the head variables, and remove duplicates.

VI. Experiments

We evaluate the performance of our solution over a co-author graph relation (`hop`) that we have extracted from http://www.dblp.org/ and compare performance to the naive method introduced in Section III-D. We have created subsets of the co-author relation with 100, 1K, 10K, 100K, and 1M tuples, respectively. All experiments were run on a machine with 2 x 3.3Ghz AMD Opteron 4238 Processors (12 cores in total) and 128GB RAM running Oracle Linux Server release 6.4. We used the commercial DBMS X (name omitted due to licensing restrictions) as a backend for our system. Unless stated otherwise, each experiment was repeated 100 times and we report the median runtime. Each run was allocated a timeslot of 10 min. Computations that did not finish in the allocated time are omitted from graphs.

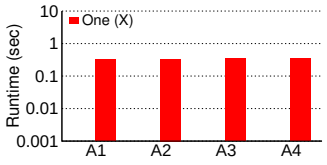


(a) `hopNotSelf` (Why) (b) `hopNotSelf` (Why-not)

(c) `only2hop` (Why) (d) `only2hop` (Why-not)

(e) `3level3hop` (Why) (f) `3level3hop` (Why-not)

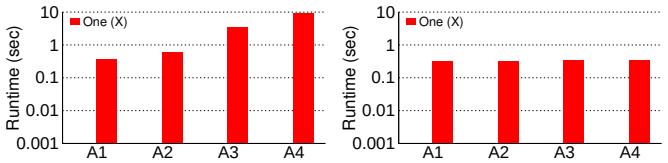
Fig. 14: Runtime - Varying Instance Size



(a) `only2hop` (Why)

Author	(a)	(b)	(c)
A1 = A. Umlauf	10	30	3
A2 = W. Hu	100	936	10
A3 = J. Davis	962	29644	30
A4 = W. Regli	1829	61417	60

(b) Output Size



(c) `3level3hop`(Why) (d) `hopNotSelf` (Why)

Fig. 15: Runtime - Varying Number of Co-authors

Workloads. We use the queries shown below: `only2hop` is our example query expressed over the co-author graph; `hopNotSelf` is a direct co-author relationship without a self-loop for the second author (those self loops do not exist in the dataset, we have added this goal to control the output size for why-not); `3level3hop` finds paths of length 3. This query is split across 3 rules to increase the complexity and size of the generated graphs (two levels of rules in the PG).

$$\texttt{only2hop}(X,Y) :- \texttt{hop}(X,Z), \texttt{hop}(Z,Y), \neg\, \texttt{hop}(X,Y)$$
$$\texttt{3level3hop}(X,Y) :- \texttt{hop}(X,A), \texttt{Q}_1(A,B), \texttt{Q}_2(B,Y)$$
$$\texttt{Q}_1(X,Y) :- \texttt{hop}(X,Y)$$
$$\texttt{Q}_2(X,Y) :- \texttt{hop}(X,Y)$$
$$\texttt{hopNotSelf}(X,Y) :- \texttt{hop}(X,Y), \neg\, \texttt{Q}'(Y).$$
$$\texttt{Q}'(Y) :- \texttt{hop}(Y,Y)$$

Naive Implementation. As mentioned in Section I-B, the naive implementation has to instantiate a game graph with $\mathcal{O}(\|adom(I)\|^k)$ where k is the maximal number variables in a rule. We do not have an implementation of the naive approach available. To compute a very conservative lower bound on the runtime of this method, we ran a query that computes an n-way cross-product over the active domain. Note that the actual runtime will be much higher, because 1) several edges are created for each rule binding and 2) several recursive datalog queries have to be evaluated over this graph. The results for different instance sizes and number of variables (k) is shown in the table below. For 2 variables, the runtime is a lower bound for the performance of the `hopNotSelf` query, for 3 variables it bounds the runtime of `only2Hop`, and for 4 variables the runtime of `2level3hop`. Even for only 2 variables, the query did not finish within the 10min limit. For 4 variables, the query only finishes within the limit for the 100 tuple instance.

Number of Vars \Instance Size	100	1K	10K	100K
w/ 2 Variables (`hopNotSelf`)	0.043	0.171	14.016	-
w/ 3 Variables (`only2hop`)	0.294	285.524	-	-
w/ 4 Variables (`3level3hop`)	56.070	-	-	-
- = did not finish in the allocated time (10 min)				

Varying Instance Size. In this experiment, we vary the instance size from 100 to 1M tuples and measure the runtime for computing explanations. As shown in Fig. 14, our approach enables us to compute provenance games for why- and why-not questions over large instances and significantly outperforms the naive implementation even for smaller instances. For `hopNotSelf` (Fig. 14.a and 14.b), binding both variables results in a unified program without variables, i.e., an existence or non-existence check. Not surprisingly, such queries are fast, even for the largest instance. If only one variable is bound, the query requires an existence (non-existence check) for any tuple (p, p) in the instance such that p is connected to the person we have bound X to. This results in linear growth in runtime based on the instance size. Note that for why-not questions, the number of rule bindings will be linear in the instance. That is, unless a compressed representation of provenance games is used, it is not possible to achieve sub-linear runtime. For the `only2hop` and `3level3hop` queries, the runtime for why questions still follows the same trend. For why-not questions, the increasing number of existential variables in the rules causes a significant increase in result size. For instance, if two variables are bound in query `only2hop`, then the resulting unified rule has one existential variable. Thus, the runtime increase in instance size is linear as expected. For one bound variable, the number of output tuples produced by the provenance computation is quadratic in the instance size resulting in quadratic increase in runtime.

Varying Output Size. We now fix the instance size and vary the query result size by carefully choosing bindings of variables to persons. For this experiment, the instance size was fixed to 100K. We only focus on why questions, since the output size for why-not only varies slightly based on which constants are used in the user question. Recall that the number of lost rule bindings is determined by the number of existential variables. The only variation is the number of lost goals. As shown in Fig. 15, the effect of the output size on runtime is dominated by other factors for `hopNotSelf` and `only2hop`. However, the performance of the 3-way join `3level3hop` is effected by the result size. Fig. 15b shows the number of results produced for the authors used as constants in the user questions.

VII. Conclusions

We present a unified framework for explaining answers and non-answers over first-order queries. Our efficient middleware implementation generates a Datalog program that computes the provenance game for a question and compiles this program into SQL code. Our experimental evaluation demonstrates that by integrating all phases of game construction and by avoiding to generate parts of the game that are irrelevant (bad moves) or unrelated to the user question we can significantly outperform the previous solution for constructing provenance games. One interesting avenue for future work is to investigate compressed and summarized representation of provenance (e.g., in the spirit of [12], [23], [5], or [10]) to deal with the large size of games for queries with negation. Other topics of interest include considering integrity constraints in the provenance game (e.g., rule instantiations can never succeed if they violate integrity constrains), marrying the approach with ideas from missing answer approaches that only return one explanation that is optimal according to some criterion, and extend it towards more expressive query languages (e.g., aggregation or non-stratified recursive programs).

References

[1] P. Alvaro, J. Rosen, and J. M. Hellerstein. Lineage-driven fault injection. In *SIGMOD*, pages 331–346, 2015.

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

[3] N. Bidoit, M. Herschel, and K. Tzompanaki. Immutably answering why-not questions for equivalent conjunctive queries. In *TaPP*, volume 1, 2014.

[4] N. Bidoit, M. Herschel, K. Tzompanaki, et al. Query-based why-not provenance with nedexplain. In *EDBT*, pages 145–156, 2014.

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

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

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

[8] C. V. Damásio, A. Analyti, and G. Antoniou. Justifications for logic programming. In *Logic Programming and Nonmonotonic Reasoning*, pages 530–542. Springer, 2013.

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

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

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

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

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

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

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

[16] M. Herschel. Wondering why data are missing from query results?: ask conseil why-not. In *CIKM*, pages 2213–2218, 2013.

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

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

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

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

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

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

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

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

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

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

APPENDIX

A. Proofs

THEOREM 1 *The graph returned by program* $\mathbb{GP}(Q(t), P)$ *over instance* I *is* $\text{EXPL}(Q(t), P, I)$.

Proof: This outline of the proof for this theorem is shown below:

1) We prove that only edges corresponding to "good" moves that occur in the instantiated game are returned by the program
2) Next we prove that the won/lost status of nodes returned by the program is correct
3) Based on the previous two steps we now know that the program returns a subset of edges of the full provenance game. It remains to be shown that this subset contains exactly the edges of the explanation $\text{EXPL}(Q(t), P, I)$.

Step 1 - Correct Edges and No Bad Moves

By construction the program cannot return bad moves (moves from a won to a won node), because there are no rules with head predicate M that create edges where both nodes are won. The constant values used as variable binding by the rules creating edges in $\mathbb{GP}(Q(t), P)$ are either constants that occur in the user question or are the result of rules which are evaluated over the instance. Since only the rules for creating the edge relation create new values (through skolem functions), it follows that any contant used in constructing a node argument exists in the active domain. Recall that the instantiated game will contain nodes with all possible argument combinations using values of the active domain. All nodes created by the program are contained in the instantiated game if we ignore their won/lost status. Any edge returned by the program is strictly based on the structure of the input program and connects nodes that agree on variable bindings. Thus, when removing the won/lost status, each edge produced by the program will be contained in the instantiated game.

Step 2 - Correct Won/Lost Status

It has been proven that the provenance game encode query evaluation in [21]. Thus, a relation node is won/lost iff it is in/not in the query result. The status of goal nodes, and negated relation nodes is completely determined by the status of the relation and rule nodes they are connected to (and whether the goal is negative or positive). Hence, to prove that the won/lost status of nodes connected by edges returned by the program $\mathbb{GP}(Q(t), P)$ is correct, we have to prove that a relation node is won/lost iff it is returned/not returned by the query.

The rules creating the edge relation of a provenance game connect a won relation node $Q(\vec{X})$ to a lost rule node $r(\vec{X}, \vec{Y})$ if the positive firing rule for $\mathtt{F}_{\mathtt{r},\mathtt{T}}$ for r returns $r(\vec{X}, \vec{Y})$. The rules creating the edge relation of a provenance game connect a lost relation node $Q(\vec{X})$ to a won rule node $r(\vec{X}, \vec{Y})$ if the negative firing rule $\mathtt{F}_{\mathtt{r},\mathtt{F}}$ for r returns $r(\vec{X}, \vec{Y}, V_1, \ldots, V_n)$ where at least one of the V_i's is false and $\neg Q(\vec{X})$ holds. Thus, as long as the firing rules correctly determine rule bindings, their won/lost state, and the won/lost state of head relation nodes, only edges occurring in the provenance game Γ can be returned.

We prove that the firing rules exhibit these properties by induction over the depth of an input program. The depth $d(P)$ of program P is defined as:

$$d(P) = \max_{r \in P} d(r)$$

$$d(r) = \max_{g \in body(r)} d(pred(g))$$

$$d(R) = \begin{cases} 0 & \text{if } R \text{ is EDB relation} \\ \max_{r \in P \wedge head(r) = R} d(r) + 1 & \text{else} \end{cases}$$

That is rules accessing only EDB predicates have depth 1 and in the general case, the depth of a rule determines how many levels of rules have to be evaluated to compute the goal predicates needed to evaluate the rule.

Base Case: Assume we have a program P with $d(P) = 0$. For a program of depth 0, all rules only access EDB predicates. We first prove that positive and negative firing rules for EDB relations are correct, because these are these rules are used in firing rules for the rules of depth 0 programs. An EDB firing rule $\mathtt{F}_{\mathtt{R},\mathtt{T}}$ creates a copy of the input relation. Thus, $t \in R$ iff $t \in \mathtt{F}_{\mathtt{R},\mathtt{T}}$. For the negative case ($\mathtt{F}_{\mathtt{R},\mathtt{F}}$), all variables are bound to the active domain and the explicitly checks that $\neg R(\vec{X})$ is true. Finally, $\mathtt{F}_{\mathtt{R},}$ uses $\mathtt{F}_{\mathtt{R},\mathtt{T}}$ and $\mathtt{F}_{\mathtt{R},\mathtt{F}}$ to determine whether or not the tuple exists. Thus, as long as these rules are correct, then it follows that $\mathtt{F}_{\mathtt{R},\mathtt{F}/\mathtt{T}}$ is correct too. For rules we have to distinguish between two cases based on the rule adornment: T or F.

For a rule $r : Q(\vec{Y}) :- g_1(\vec{X_1}), \ldots, g_n(\vec{X_n})$ with $vars(r) = \vec{X}$ adorned with T, there are two firing rules: $\mathtt{F}_{\mathtt{Q},\mathtt{T}}(\vec{Y}) :- \mathtt{F}_{\mathtt{r},\mathtt{T}}(\vec{X})$ and $\mathtt{F}_{\mathtt{r},\mathtt{T}}(\vec{X}) :- g_1(\vec{X_1}), \ldots, g_n(\vec{X_n})$. Rule $\mathtt{F}_{\mathtt{r},\mathtt{T}}$ and r have the same body and, thus, a rule instantiation for r is successful iff $\mathtt{F}_{\mathtt{r},\mathtt{T}}$ is successful. Rule $\mathtt{F}_{\mathtt{r},\mathtt{T}}$ returns all variables bound in its body. Thus, each tuple in $\mathtt{F}_{\mathtt{r},\mathtt{T}}$ corresponds to successful instantiation of the variables in the body of rule r. The same argument holds for each rule r' with $head(r) = Q$. It follows that $t \in Q \Leftrightarrow t \in \mathtt{F}_{\mathtt{r},\mathtt{T}}$.

For a rule $r : Q(\vec{Y}) :- g_1(\vec{X_1}), \ldots, g_n(\vec{X_n})$ with $vars(r) = \vec{X}$ adorned with F, the body contains an additional goal $\neg Q(\vec{y})$. Thus, this rule can only return bindings for lost relation nodes (if a tuple is missing then all rule bindings that would have produced this tuple must have failed). Except for this additional goal, the firing rule operates in the same fashion as in the positive case and, thus, is proven correct through the same argument. Since $\mathtt{F}_{\mathtt{R},\mathtt{F}/\mathtt{T}}$ was proven correct, it follows that the additional head variables V_i correctly record whether the goal is successful or not. For the head predicate of a rule adorned with F we create two firing rules. Rule $\mathtt{F}_{\mathtt{Q},\mathtt{T}}$ is the same as before (and hence is correct). Rule $\mathtt{F}_{\mathtt{Q},\mathtt{F}}$ binds variables to the active domain and checks $\neg \mathtt{F}_{\mathtt{Q},\mathtt{T}}(\vec{Y})$. This rule obviously computes the complement of Q as expected.

Inductive Step: Assume that firing rules for programs of depth up to n are correct. We need to prove that firing rules for programs of depth of up to $n+1$ are correct. From the assumption it follows that all firing rules for rules with a depth of at most $n-1$ are correct. It remains to show that firing rules for rules of depth $n+1$ in such a program are correct. Note that in the arguments made for rules in the base case, the fact that firing rules for the goals in the body correspond to EDB

relation is immaterial to the argument. The only assumption was that these firing rules have been proven to be correct. Since, we know from the induction hypothesis that all firing rules for EDB relations and relations that only occur as head predicates of rules with depth less than $n-1$, it follows that the same argument can be applied to prove that firing rules for rules of depth $n+1$ are correct.

Step 3 - $\mathbb{GP}(Q(t), P)$ Returns Explanation for $Q(t)$

Having established that edges returned by the program $\mathbb{GP}(Q(t), P)$ are contained in the provenance game Γ for P and I, we now have to prove that this set of edges form precisely $\text{EXPL}(Q(t), P, I)$. We prove this fact by induction over the depth of a program. We prove this fact for a user question containing only constants. The extension to user questions which contain variables is immediate.

Base Case: Consider a program of depth 0. For a rule node to be connected to the user question $Q(t)$, it's head variables have to be bound to t. This is exactly the restriction applied by the unification step. Since, the firing rules are known to be correct, this guarantees that exactly the rules nodes connected to the user question are generated. The propagation of this unification to the firing rules for EDB relations is correct, because only EDB relation nodes with these bindings can be connected to rules nodes (after all only nodes for which constants agree are connected). However, propagating constants is not sufficient. The firing rules for EDB predicates may return tuples that correspond to relation nodes that are not connected to the rules nodes connected to the user question. Consider an EDB firing rule $\mathtt{F}_{\mathtt{R},\sigma}$ (here σ denotes any adornment, we ignore adornments since they are irrelevant for this part of the argument) and a rule firing rule $\mathtt{F}_{\mathtt{r},\sigma}(\vec{Y})$ for a rule $r : Q(\vec{X}) :- \ldots R(\vec{X'}) \ldots$ with R as the i^{th} goal. A node $R(t)$ would be connected to $r(t')$ iff $\vec{X'}$ is bound to the same constants as $\vec{Y}$. This is checked in the firing rule $\mathtt{FC}_{\mathtt{R},\mathtt{r}^\mathtt{i},\sigma}$ with connectivity join by unifying $\vec{Y}$ and $\vec{X'}$. Thus, if tuple is returned by such a connected firing rule, its node is guaranteed to be connected to the rule node and, thus, also the user question.

Inductive Step: Assume that the hypothesis holds for program of depth up to n. We have to show that this also holds for programs of depth $n+1$. For any rules of depth 1 or larger, correctness follows from the induction hypothesis. followsIt remains to be shown that this implies correctness for rules of depth 0. Note that the argument applied in the base case does not rely on the fact that $Q(t)$ is the user question. If we replace $Q(t)$ with an IDB relation node that is connected to $Q(t)$, then the argument still holds. Thus, we can apply this argument to prove that the rules of depth 0 in a program of depth $n+1$ are correct. ■

LEMMA 1 *Let $Q(t)$ be a user question over program P and instance I. The size of the program P_U produced by step 1 of our algorithm may be exponentially larger than P.*

Proof: Consider the following class of programs P_n where each program P_n in the program has n^2+n rules:

$$
\begin{aligned}
&Q_1(X_1,\ldots,X_n) :- Q_2(X_1,\ldots,X_n)\\
&Q_1(X_1,\ldots,X_n) :- Q_2(X_2,X_1,\ldots,X_n)\\
&Q_1(X_1,\ldots,X_n) :- Q_2(X_3,X_2,X_1,\ldots,X_n)\\
&\ldots\\
&Q_1(X_1,\ldots,X_n) :- Q_2(X_n,X_2,\ldots,X_1)\\
\\
&Q_2(X_1,\ldots,X_n) :- Q_3(X_1,\ldots,X_n)\\
&Q_2(X_1,\ldots,X_n) :- Q_3(X_2,X_1,\ldots,X_n)\\
&Q_2(X_1,\ldots,X_n) :- Q_3(X_3,X_2,X_1,\ldots,X_n)\\
&\ldots\\
&Q_2(X_1,\ldots,X_n) :- Q_3(X_n,X_2,\ldots,X_1)\\
\\
&\ldots\\
&Q_{n+1}(X_1,\ldots,X_n) :- Q_n(X_n,X_2,\ldots,X_1)
\end{aligned}
$$

A program P_n has n IDB predicates with i rules with i variables each. Consider how the constants for a question $Q_1(t)$ with $t = (1,\ldots,n)$ would be propagated through the program. Each of the rules for a predicate Q_i create a permutation of the constants by switching X_1 with any other X_i. For instance, the first rule switches X_1 with itself. Thus, the unification process will create n bindings for Q_2, e.g., $(2,1,\ldots,n)$ and $(3,2,1,\ldots,n)$. Note that after $n+1$ steps any permutation of $(1,\ldots,n)$ would have been created. Since there are $n!$ permutations of n numbers, the number of unified rules created for Q_n will be $n!$. To see why it is true that all permutations will be created, consider how $(1,\ldots,n)$ can be permuted into $(i_1,\ldots,i_n)$ by switching an element with the current first element $n+1$ times. Let pos_j denote the position of element j in $(i_1,\ldots,i_n)$. We repeatedly switch the element j currently at position 1 with the element currently at position pos_j. The exception is the element k with $pos_k = 1$. Unless we are in the last step of the permutation, this element will be switched with an element l that is not currently at pos_l. The element k will then be switched into the current position of l. If l is chosen carefully, then we will be able to switch k back to position 1 in the last step. ■

Note that we can avoid this blowup by simply removing the unification step. However, the unification step can provide useful selection conditions for why questions over positive programs and can reduce the exponential factor when computing why-not questions. In practise, we have yet to see any program where this step results in a super-linear increase in program size. Thus, for now we will assume that the increase in program size caused by the unification step is bounded by a constant factor. If this should ever become a concern, we could implement a safety check that stops unification if the increase in program size exceeds a threshold.

B. Algorithm Pseudocode

Algorithm 6 Create Edge Relation

procedure CREATEEDGERELATION(P_{FC}, $Q(t)$)
  $P_M \leftarrow []$
  $todo \leftarrow [Q(t)]$
  $done \leftarrow \{\}$
  while $todo \neq []$ **do**
    $R(t)^\sigma \leftarrow$ POP($todo$)
    if $R(t)^\sigma \in done$ **then**
      continue
    $done \leftarrow$ INSERT($done, R(t)^\sigma$)
    $rules \leftarrow$ GETRULES($R(t)^\sigma$)
    for all $r \in rules$ **do**
      if ISEDB(R) **then**
        if $\sigma = T$ **then**
          $r_{new} \leftarrow \texttt{edge}(f_R^T(t), f_{r_R}^F(t)) :- \texttt{FC}_{\texttt{R},\texttt{r}^\texttt{i},\texttt{T}}(t)$
          $P_M \leftarrow P_M :: r_{new}$
      else
        $\sigma_r =$ SWITCHSTATE(σ)
        $r_{new} \leftarrow \texttt{edge}(f_{pred(r)}^\sigma(t), f_r^{\sigma_r}(t, \ldots)) :- \texttt{F}_{\texttt{r},\sigma}(t, \ldots)$
        $P_M \leftarrow P_M :: r_{new}$
        for all $g(t') \in body(r)$ **do**
          if ISNEGATED(g) **then**
            $\sigma' \leftarrow$ SWITCHSTATE(σ)
          else
            $\sigma' \leftarrow \sigma$
          $todo \leftarrow todo :: g(t')^{\sigma'}$
          if $\sigma = T$ **then**
            $r_{r\to g} \leftarrow \texttt{edge}(f_r^F(args), f_g^T(t')) :- \texttt{F}_{\texttt{r},\texttt{T}}(args)$
            if ISNEGATED(g) **then**
              $r_{g\to R} \leftarrow \texttt{edge}(f_g^T(t'), f_R^F(t')) :- \texttt{F}_{\texttt{r},\texttt{T}}(args)$
              $P_M \leftarrow P_M :: r_{r\to g} :: r_{g\to R}$
            else
              $r_{g\to\neg R} \leftarrow \texttt{edge}(f_g^T(t'), f_{\neg R}^F(t')) :- \texttt{F}_{\texttt{r},\texttt{T}}(args)$
              $r_{\neg R\to R} \leftarrow \texttt{edge}(f_{\neg R}^F(t'), f_R^T(t')) :- \texttt{F}_{\texttt{r},\texttt{T}}(args)$
              $P_M \leftarrow P_M :: r_{r\to g} :: r_{g\to\neg R} :: r_{\neg R\to R}$
          else
            $args_b \leftarrow b_1, \ldots, b_{i-1}, false, b_{i+1}, \ldots, b_n$
            $r_{r\to g} \leftarrow \texttt{edge}(f_r^T(args), f_g^F(t')) :- \texttt{F}_{\texttt{r},\texttt{T}}(args, args_b)$
            if ISNEGATED(g) **then**
              $r_{g\to R} \leftarrow \texttt{edge}(f_g^F(t'), f_R^T(t')) :- \texttt{F}_{\texttt{r},\texttt{T}}(args, args_b)$
              $P_M \leftarrow P_M :: r_{r\to g} :: r_{g\to R}$
            else
              $r_{g\to\neg R} \leftarrow \texttt{edge}(f_g^F(t'), f_{\neg R}^T(t')) :- \texttt{F}_{\texttt{r},\texttt{T}}(args, args_b)$
              $r_{\neg R\to R} \leftarrow \texttt{edge}(f_{\neg R}^T(t'), f_R^F(t')) :- \texttt{F}_{\texttt{r},\texttt{T}}(args, args_b)$
              $P_M \leftarrow P_M :: r_{r\to g} :: r_{g\to\neg R} :: r_{\neg R\to R}$
  return P_M