Implementing Unified Why- and Why-Not Provenance through Games

Seokki Lee[1], Sven Köhler[2], Bertram Ludäscher[3], and Boris Glavic[1]

[1] Illinois Institute of Technology
`slee195@hawk.iit.edu, bglavic@iit.edu`
[2] University of California, Davis
`svkoehler@ucdavis.edu`
[3] University of Illinois at Urbana-Champaign
`ludaesch@illinois.edu`

Abstract. Using provenance to explain *why* a query returns a result or why a result is *missing* has been studied extensively. However, the two types of questions have been approached independently of each other. We present an efficient technique for answering both types of questions for Datalog queries based on a game-theoretic model of provenance called *provenance games*. Our approach compiles provenance requests into Datalog and translates the resulting query into SQL to execute it on a relational database backend. We apply several novel optimizations to limit the computation to provenance relevant to a given user question.

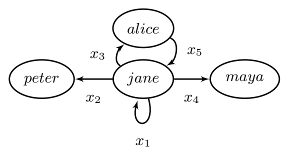
1 Introduction

Explaining the existence and absence of query results through *provenance* respective *missing answer* techniques can help users to, e.g., debug and understand their data and queries. Recently, the two techniques have been unified [3] in a single framework based on a game-theoretic notion of provenance for queries with negation, particularly, for non-recursive Datalog$^{\neg}$. [4] The provenance game for a query Q and database instance I explains for each existing and missing query result how the rules of the query succeeded (respective failed) to derive it and why the derivation succeeded (respective failed), i.e., which tuples present or absent in the database instance caused rule derivations to succeed (respective fail). Typically, a user would not be interested in explanations for all answers and non-answers, but rather would like to understand why a particular tuple is (not) in the result. Given such a user question $\mathtt{Q}(t)$, our approach computes a subgraph of the full game that answers precisely the user question. While provenance games provide a solid underlying theoretical foundation, these games are not necessarily the most user-friendly representation of provenance, i.e., they require some background in game theory to be interpreted correctly. Our system

[4] Intuitively, asking why a tuple t is absent from Q is equivalent to explaining why t is present in $\neg Q$. Thus, a provenance model with support for negation in queries enables *why* and *why-not* questions to be treated uniformly.

Relation **L**

S	T	
jane	jane	x_1
jane	peter	x_2
jane	alice	x_3
jane	maya	x_4
alice	jane	x_5



Result of Query **Q**

S
jane
alice

Fig. 1: Example database

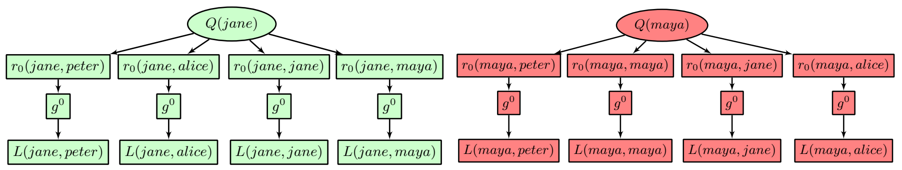


Fig. 2: Provenance explaining why Q(jane) and why-not Q(maya).

also supports several simpler forms of provenance that can be derived from a provenance game by graph transformations, e.g., we support graphs that encode provenance polynomials [2] for positive queries. Importantly, the core of our technique is independent of how provenance is represented eventually and, thus, new types of provenance representations can be added easily. The conventional method [3] for computing provenance games is not suited well for computing the part of the provenance game explaining a single answer or non-answer $\mathtt{Q}(t)$, because it has to instantiate the full game which is prohibitively expensive, even for small instances. For example, for a database with 1000 values and a query with a single rule using 5 variables, the full game will contain more than 10^{15} nodes. Our approach computes the provenance bottom-up and only instantiates parts of the game if they may be relevant to answer the user question.

Example 1 *Consider relation* L *in Fig. 1, which stores links between personal webpages. For example, the tuple* (jane,peter) *denotes that Jane's webpage contains a link to Peter's webpage. A webpage may contain links to other parts of the page (a self-loop). Consider a query Q expressed in Datalog that returns webpages that have outgoing links:* $r_0 : \mathtt{Q}(X) :\!- \mathtt{L}(X, Y)$. *Given such a query, a user may be interested in understanding* why *or* why-not *a webpage occurs in the result of query Q. Fig. 2 shows the simplified provenance graphs produced by our approach for several why- and why-not questions. For instance, tuple* (*jane*) *is in the result (denoted by the green background), because there is a link from her webpage to Alice's (tuple* x_3*) which causes rule derivation* $r_0(\mathit{jane}, \mathit{alice})$ *to succeed. Tuple* (*maya*) *is not in the result, because none of the four possible links connecting her webpage to any of the other webpages in the database exists. Thus all possible derivations of* Q(*maya*) *using rule* r_0 *have failed.*

2 Efficiently Generating Provenance Games

The input to our approach for computing provenance games is a Datalog program and either a *why* or *why-not* question, i.e., why is tuple t in the result respective missing from the result. Furthermore, the user can select whether one of the simplified provenance representations should be returned. Based on these inputs, we construct a new Datalog program that computes the edge relation of the provenance game graph for t as detailed in the following.

1) Unify program with provenance request. We first unify the program with the question $\mathtt{Q}(t)$ by propagating the constants in t to replace variables throughout the program in order to limit the computation to relevant parts of the game. For example, to explain why $\mathtt{Q}(\mathsf{jane})$ (on the left in Fig. 2), we only have to consider rule bindings where $X = \mathsf{jane}$.

2) Annotated rules. We then determine for which nodes in the graph we can infer their success/failure state based on the user question. For instance, we only need to consider successful instantiations of rule r_0 to explain why $\mathtt{Q}(\mathsf{jane})$. We store this information as annotations on rules and goals in the Datalog program.

3) Capture rule derivations. Based on the annotated and unified game created in the previous steps, we generate rules capturing variable bindings for successful and failed rule instantiations (the annotations enable us to determine whether we can focus on successful or failed instantiations only) in order to construct the subgraph of a provenance game corresponding to a rule derivation. We call these rules *firing rules*.

Successful derivations. Reconsider question why $\mathtt{Q}(\mathsf{jane})$ from Example 1. The firing rule capturing successful bindings of r_0, the only rule of query Q, is derived from r_0 by adding Y (the only existential variable in r_0) to the head, renaming the head predicate as $\mathtt{F}_{\mathtt{r_0},\mathtt{T}}$, and replacing each goal with its firing version. Firing rules are created after the unification with the user question. Thus, for the example question, we would start from $r_0 : \mathtt{Q}(\mathsf{jane}, Y) :- \mathtt{L}(\mathsf{jane}, Y)$. Positive firing rules for edb predicates simply copy the predicate.

$$\mathtt{F}_{\mathtt{r_0},\mathtt{T}}(\mathsf{jane}, Y) :- \mathtt{F}_{\mathtt{L},\mathtt{T}}(\mathsf{jane}, Y) \qquad \mathtt{F}_{\mathtt{L},\mathtt{T}}(\mathsf{jane}, Y) :- \mathtt{L}(\mathsf{jane}, Y)$$

Failed derivations. To construct a provenance graph fragment corresponding to a missing tuple, we find failed derivations with this tuple in the head and ensure that no successful derivations of the tuple exist (otherwise we may capture the irrelevant failed derivations of existing tuples). Furthermore, we need to determine which goals failed for each failed rule instantiation because only failed goals will be connected to the failed rule instantiations in the provenance game. For the why-not question $\mathtt{Q}(\mathsf{maya})$ shown in Fig. 2 (on the right side), we are only interested in failed instantiations of rule r_0 with $X = \mathsf{maya}$. The generated firing rules are shown in Fig. 3. A negative firing rule (capturing failed derivations) is constructed by replacing every goal in the body with its F/T firing version. An F/T firing rule captures both existing and missing tuples and uses an additional boolean variable (V_1 in Fig. 3) in the head to record whether a tuple is existing or missing. We also add a firing rule for the negated head atom to the body to only capture bindings for missing tuples. Since query Q (in the Example 1) has only one goal, we simply capture whether this goal is won or lost for each rule instantiation using boolean variable V_1. As mentioned above, we use a F/T firing rule for relation $\mathtt{L}$ to determine whether a tuple exists in $\mathtt{L}$.

$$
\begin{aligned}
\mathtt{F}_{\mathtt{Q},\mathtt{F}}(\mathsf{maya}) &:- \neg\, \mathtt{F}_{\mathtt{Q},\mathtt{T}}(\mathsf{maya})\\
\mathtt{F}_{\mathtt{Q},\mathtt{T}}(\mathsf{maya}) &:- \mathtt{F}_{\mathtt{r_0},\mathtt{T}}(\mathsf{maya}, Y)\\
\mathtt{F}_{\mathtt{r_0},\mathtt{F}}(\mathsf{maya}, Y, V_1) &:- \mathtt{F}_{\mathtt{Q},\mathtt{F}}(\mathsf{maya}),\\
&\qquad \mathtt{F}_{\mathtt{L},\mathtt{F}/\mathtt{T}}(\mathsf{maya}, Y, V_1)\\
\mathtt{F}_{\mathtt{r_0},\mathtt{T}}(\mathsf{maya}, Y) &:- \mathtt{F}_{\mathtt{L},\mathtt{F}/\mathtt{T}}(\mathsf{maya}, Y, true)\\
\mathtt{F}_{\mathtt{L},\mathtt{F}/\mathtt{T}}(\mathsf{maya}, Y, true) &:- \mathtt{L}(\mathsf{maya}, Y)\\
\mathtt{F}_{\mathtt{L},\mathtt{F}/\mathtt{T}}(\mathsf{maya}, Y, false) &:- \mathtt{adom}(Y), \neg\, \mathtt{L}(\mathsf{maya}, Y)
\end{aligned}
$$

Fig. 3: Firing rules for failed derivations

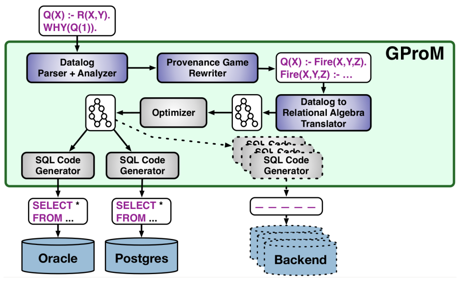


Fig. 4: GProM Implementation

4) Filter out false positives. To be in the result of one of the firing rules obtained in the previous step is a necessary, but not sufficient condition for the provenance graph fragment corresponding to this rule binding to be connected to the user question. To guarantee that only relevant fragments are returned, we need to check for each fragment whether it is actually connected. We introduce additional rules that check connectivity one hop at a time.

5) Compute edge relation. We compute the edge relation of the provenance game based on the rule binding information that the firing rules have captured. In addition to full game provenance, we support simplified provenance representations including the ones shown in Fig. 2.

Implementation. The generated Datalog program constructs and solves the provenance game simultaneously in a bottom-up manner. We have implemented this algorithm in our provenance middleware called GProM [1] that executes provenance requests using a database backend. The process of computing a provenance game for a user request is shown in Fig. 4. Our system also visualizes the resulting graph using Graphviz (`http://www.graphviz.org/`).

3 Conclusions

We present an efficient approach for explaining answers and non-answers to Datalog queries using provenance games. Our approach limits the computation to parts of the provenance relevant to a user question by constructing the game bottom-up and pruning unrelated parts from the computation.

References

1. Arab, B., Gawlick, D., Radhakrishnan, V., Guo, H., Glavic, B.: A generic provenance middleware for database queries, updates, and transactions. In: TaPP (2014)
2. Green, T., Karvounarakis, G., Tannen, V.: Provenance semirings. In: PODS. pp. 31–40 (2007)
3. Köhler, S., Ludäscher, B., Zinn, D.: First-order provenance games. In: In Search of Elegance in the Theory and Practice of Computation, pp. 382–399. Springer (2013)