The Complexity of Causality and Responsibility for Query Answers and non-Answers*

Alexandra Meliou Wolfgang Gatterbauer Katherine F. Moore Dan Suciu
Department of Computer Science and Engineering,
University of Washington, Seattle, WA, USA
{ameli,gatter,kfm,suciu}@cs.washington.edu

ABSTRACT

An answer to a query has a well-defined lineage expression (alternatively called how-provenance) that explains how the answer was derived. Recent work has also shown how to compute the lineage of a non-answer to a query. However, the *cause* of an answer or non-answer is a more subtle notion and consists, in general, of only a fragment of the lineage. In this paper, we adapt Halpern, Pearl, and Chockler's recent definitions of causality and responsibility to define the *causes of answers and non-answers to queries, and their degree of responsibility*. Responsibility captures the notion of degree of causality and serves to rank potentially many causes by their relative contributions to the effect. Then, we study the complexity of computing causes and responsibilities for conjunctive queries. It is known that computing causes is NP-complete in general. Our first main result shows that all causes to conjunctive queries can be computed by a relational query which may involve negation. Thus, causality can be computed in PTIME, and very efficiently so. Next, we study computing responsibility. Here, we prove that the complexity depends on the conjunctive query and demonstrate a dichotomy between PTIME and NP-complete cases. For the PTIME cases, we give a non-trivial algorithm, consisting of a reduction to the max-flow computation problem. Finally, we prove that, even when it is in PTIME, responsibility is complete for LOGSPACE, implying that, unlike causality, it cannot be computed by a relational query.

1. INTRODUCTION

When analyzing complex data sets, users are often interested in the *reasons* for surprising observations. In a database context, they would like to find the *causes of answers or non-answers* to their queries. For example, "What caused my personalized newscast to have more than 50 items today?" Or, "What caused my favorite undergrad student to not appear on the Dean's list this year?" Philosophers have debated for centuries various notions of causality, and today it is still studied in philosophy, AI, and cognitive science.

*This work was partially funded by NSF IIS-0911036, IIS-0915054, and IIS-0713576. For more information see the project webpage: http://db.cs.washington.edu/causality/

Proceedings of the VLDB Endowment, Vol. 4, No. 1

Database schema

Director(*did*, *firstName*, *lastName*)
Movie(*mid*, *name*, *year*, *rank*)
Movie_Directors(*did*, *mid*)
Genre(*mid*, *genre*)

Query

```
SELECT    DISTINCT g.genre
FROM      Director d, Movie_Directors md,
          Movie m, Genre g
WHERE     d.lastName LIKE 'Burton'
AND       g.mid=m.mid
AND       m.mid=md.mid
AND       md.did=d.did
ORDER BY  g.genre
```

Query answers

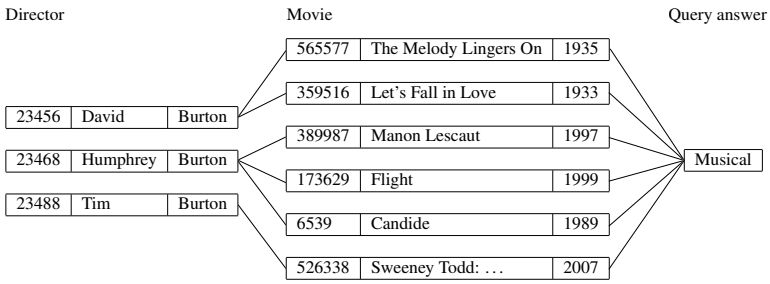
genre
...
Drama
Family
Fantasy
History
Horror
Music
Musical
Mystery
Romance
Sci-Fi
...

Figure 1: A SQL query returning the genres of all movies directed by Burton, on the IMDB dataset (www.imdb.org). The famous director Tim Burton is known for dark, gothic themes, so the genres Fantasy and Horror are expected. But the genres Music and Musical are quite surprising. The goal of this paper is to find the *causes for surprising query results*.

Understanding causality in a broad sense is of vital practical importance, for example in determining legal responsibility in multi-car accidents, in diagnosing malfunction of complex systems, or in scientific inquiry. A formal, mathematical study of causality was initiated by the recent work of Halpern and Pearl [13] and Chockler and Halpern [5], who gave mathematical definitions of *causality* and its related notion of *degree of responsibility*. These formal definitions lead to applications in knowledge representation and model checking [5, 9, 10]. In this paper, we adapt the notions of causality and responsibility to database queries, and study the complexity of computing the causes and their responsibilities for answers and non-answers to conjunctive queries.

EXAMPLE 1.1 (IMDB). *Tim Burton is an Oscar nominated director whose movies often include fantasy elements and dark, gothic themes. Examples of his work are "Edward Scissorhands", "Beetlejuice" and the recent "Alice in Wonderland". A user wishes to learn more about Burton's movies and queries the IMDB dataset to find out all genres of movies that he has directed (see Fig. 1). Fantasy and Horror are quite expected categories. But Music and Musical are surprising. The user wishes to know the reason for these answers. Examining the lineage of a surprising answer is a first step towards finding its reason, but it is not sufficient: the combined lineage of the two categories consists of a total of 137 base tuples, which is overwhelming to the user.*

Causality is related to provenance, yet it is a more refined notion: Causality can answer questions like the one in our example by returning the causes of query results ranked by their *degree of responsibility*. Our starting point is Halpern and Pearl's definition



(a) Lineage from `Directors` and `Movies` of the Musical tuple

ρ_t	Answer tuple
0.33	Movie(526338, "Sweeney Todd", 2007)
0.33	Director(23456, David, Burton)
0.33	Director(23468, Humphrey, Burton)
0.33	Director(23488, Tim, Burton)
0.25	Movie(359516, "Let's Fall in Love", 1933)
0.25	Movie(565577, "The Melody Lingers On", 1935)
0.20	Movie(6539, "Candide", 1989)
0.20	Movie(173629, "Flight", 1999)
0.20	Movie(389987, "Manon Lescaut", 1997)

(b) Responsibility rankings for Musical

Figure 2: Lineage (a) and causes with their responsibilities (b) for the `Musical` tuple in Example 1.1.

of causality [13], from which we borrow three important concepts:

(1) Partitioning of variables into *exogenous* and *endogenous*: Exogenous variables define a context determined by external, unconcerned factors, deemed not to be possible causes, while endogenous variables are the ones judged to affect the outcome and are thus potential causes. In a database setting, variables are tuples in the database, and the first step is to partition them into exogenous and endogenous. For example, we may consider `Director` and `Movie` tuples as endogenous and all others as exogenous. The classification into endogenous/exogenous is application-dependent, and may even be chosen by the user at query time. For example, if erroneous data in the directors table is suspected, then only `Director` may be declared endogenous; alternatively, the user may choose only `Movie` tuples with `year`$>$`2008` to be endogenous, for example in order to find recent, or under production movies that may explain the surprising outputs to the query. Thus, the partition into endogenous and exogenous tuples is not restricted to entire relations. As a default, the user may start by declaring all tuples in the database as endogenous, then narrow down.

(2) *Contingencies*: an endogenous tuple t is a cause for the observed outcome only if there is a hypothetical setting of the other endogenous variables under which the addition/removal of t causes the observed outcome to change. Therefore, in order to check that a tuple t is a cause for a query answer, one has to find a set of endogenous tuples (called *contingency*) to remove from (or add to) the database, such that the tuple t immediately affects the answer in the new state of the database. In theory, in order to compute the contingency one has to iterate over subsets of endogenous tuples. Not surprisingly, checking causality is NP-complete in general [9]. However, the first main result in this paper is to show that the causality of conjunctive queries can be determined in PTIME, and furthermore, all causes can be computed by a relational query.

(3) *Responsibility*, a notion first defined in [5], measures the degree of causality as a function of the size of the smallest contingency set. In applications involving large datasets, it is critical to rank the candidate causes by their responsibility, because answers to complex queries may have large lineages and large numbers of candidate causes. In theory, in order to compute the responsibility one has to iterate over all contingency sets: not surprisingly, computing responsibility in general is hard for $FP^{\Sigma_2^P(\log n)}$ [5].[1] However, our second main result, and at the same time the strongest result of this paper, is a *dichotomy result for conjunctive queries*: for *each* query without self-joins, either its responsibility can be computed in PTIME in the size of the database (using a non-obvious algorithm), or checking if it has a responsibility below a given value is NP-hard.

[1]This is the class of functions computable by a poly-time Turing machine which makes $\log n$ queries to a Σ_2^p oracle.

EXAMPLE 1.2 (IMDB CONTINUED). *Continuing Example 1.1, we show in Fig. 2b the causes for Musical ranked by their responsibility score. (We explain in Sect. 2 how these scores are computed.) At the top of the list is the movie "Sweeney Todd", which* is*, indeed, the one and single musical movie directed by Tim Burton. Thus, this tuple represents a surprising fact in the data of great interest to the user. The next three tuples in the list are directors, whose last name is Burton. These tuples too are of high interest to the user because they indicate that the query was ambiguous. Equally interesting is to look at the bottom of the ranked list. The movie "Manon Lescaut" is made by Humphrey Burton, a far less known director specialized in musicals. Clearly, the movie itself is not an interesting explanation to the user; the interesting explanation is the director, showing that he happens to have the same last name, and indeed, the director is ranked higher while the movie is (correctly) ranked lower. In our simple example Musical has a small lineage, consisting of only ten tuples. More typically, the lineage can be much larger (Music has a lineage with 127 tuples), and it is critical to rank the potential causes by their degree of responsibility.*

We start by adapting the Halpern and Pearl definition of causality (HP from now on) to database queries, based on contingency sets. We define causality and responsibility both for Why-So queries ("why did the query return this answer?") and for Why-No queries ("why did the query not return this answer?"). We then prove two fundamental results. First, we show that computing the causes to any conjunctive query can be done in PTIME in the size of the database, i.e. query causality has PTIME data complexity; by contrast, causality of arbitrary Boolean expressions is NP-complete [9]. In fact we prove something stronger: the set of all causes can be retrieved by a query expressed in First Order Logic (FO). This has important practical consequences, because it means that one can retrieve all causes to a conjunctive query by simply running a certain SQL query. In general, the latter cannot be a conjunctive query, but must have one level of negation. However, we show that if the user query has no self joins *and* every table is either entirely endogenous or entirely exogenous, then the Why-So causes can be retrieved by some conjunctive query. These results are summarized in Fig. 3.

Second, we give a dichotomy theorem for query responsibility. This is our strongest technical result in this paper. For every conjunctive query without self-joins, one of the following holds: either the responsibility can be computed in PTIME or it is provably NP-hard. In the first case, we give a quite non-obvious algorithm for computing the degrees of responsibility using FordFulkerson's max flow algorithm. We further show that one can distinguish between the two cases by checking a property of the query expression that we call *linearity*. We also discuss conjunctive queries with self-joins, and finally show that, in the case of Why-No causality, one can always compute responsibility in PTIME. These results are also summarized in Fig. 3.

Causality	*Why So?*	*Why No?*
w/o SJ	PTIME (CQ)	PTIME (FO)
with SJ	PTIME (FO)	

Responsibility		*Why So?*	*Why No?*
w/o SJ	*linear*	PTIME	PTIME
	non-linear	NP-hard	
with SJ		NP-hard	

Figure 3: Complexity of determining causality and responsibility for conjunctive queries. For queries with no self-joins we provide a complete dichotomy result. Queries with self-joins are NP-hard in general, but a similar dichotomy is not known.

Causality and provenance: Causality is related to *lineage of query results*, such as why-provenance [7] or where-provenance [2]. Recently, even explanations for non-answers have been described in terms of lineage [15, 3]. We make use of this prior work because the first step in computing causes and responsibilities is to determine the lineage of an answer or non-answer to a query. We note, however, that computing the lineage of an answer is only the first step, and is not sufficient for determining causality: causality needs to be established through a contingency set, and is also accompanied by a degree (the responsibility), which are both more difficult to compute than the lineage.

Contributions and outline. Our three main contributions are:

- We define Why-So and Why-No causality and responsibility for conjunctive database queries (Sect. 2).
- We prove that causality has PTIME data complexity for conjunctive queries (Sect. 3).
- We prove a dichotomy theorem for responsibility and conjunctive queries (Sect. 4).

We review related work (Sect. 5) before we conclude (Sect. 6). All proofs are provided in the Appendix.

2. QUERY CAUSE AND RESPONSIBILITY

We assume a standard relational schema with relation names $R_1, \ldots, R_k$. We write D for a database instance and q for a query. We consider only conjunctive queries, unless otherwise stated. A subset of tuples $D^{\mathrm{n}} \subseteq D$ represents *endogenous tuples*; the complement $D^{\mathrm{x}} = D - D^{\mathrm{n}}$ is called the set of *exogenous tuples*. For each relation R_i, we write R_i^{n} and R_i^{x} to denote the endogenous and exogenous tuples in R_i respectively. If $\bar{a}$ is a tuple with the same arity as the query's answer, then we write $D \models q(\bar{a})$ when $\bar{a}$ is an answer to q on D, and write $D \not\models q(\bar{a})$ when $\bar{a}$ is a non-answer to q on D.

DEFINITION 2.1 (CAUSALITY). *Let $t \in D^{\mathrm{n}}$ be an endogenous tuple, and $\bar{a}$ a possible answer for q.*

- *t is called a* counterfactual cause *for $\bar{a}$ in D if $D \models q(\bar{a})$ and $D - \{t\} \not\models q(\bar{a})$*
- *$t \in D$ is called an* actual cause *for $\bar{a}$ if there exists a set $\Gamma \subseteq D^{\mathrm{n}}$ called a contingency for t, such that t is a counterfactual cause for $\bar{a}$ in $D - \Gamma$.*

A tuple t is a counterfactual cause, if by removing it from the database, we remove $\bar{a}$ from the answer. The tuple is an actual cause if one can find a contingency under which it becomes a counterfactual cause: more precisely, one has to find a set Γ such that, after removing Γ from the database we bring it to a state where removing/inserting t causes $\bar{a}$ to switch between an answer and a non-answer. Obviously, every counterfactual cause is also an actual cause, by taking $\Gamma = \emptyset$. The definition of causality extends naturally to the case when the query q is Boolean: in that case, a counterfactual cause is a tuple that, when removed, determines q to become false.

EXAMPLE 2.2. *Consider the query $q(x) :- R(x, y), S(y)$ on the database instance shown below, and assume all tuples are endogenous: $R = R^{\mathrm{n}}$, $S = S^{\mathrm{n}}$. Consider the answer a_2. The tuple $S(a_1)$ is a counterfactual cause for this result, because if we remove this tuple from S then a_2 is no longer an answer. Now consider the answer a_4. Tuple $S(a_3)$ is not a counterfactual cause: if we remove it from S, a_4 is still an answer. But $S(a_3)$ is an actual cause with contingency $\{S(a_2)\}$: once we remove $S(a_2)$ we reach a state where a_4 is still an answer, but further removing $S(a_3)$ makes a_4 a non-answer.*

R

X	Y
a_1	a_5
a_2	a_1
a_3	a_3
a_4	a_3
a_4	a_2

S

Y
a_1
a_2
a_3
a_4
a_6

$q(x) :- R(x, y)S(y)$

X
a_2
a_3
a_4

For a more subtle example, consider the Boolean query $q :- R(x, a_3), S(a_3)$ (where a_3 is a constant), which is true on the given instance. Suppose only the first three tuples in R are endogenous, and the last two are exogenous: $R^{\mathrm{x}} = \{(a_4, a_3), (a_4, a_2)\}$. Let's examine whether $R^{\mathrm{n}}(a_3, a_3)$ is a cause for the query being true. This tuple is not an actual cause. This is because $\{S^{\mathrm{n}}(a_3)\}$ is not a contingency for $R^{\mathrm{n}}(a_3, a_3)$: by removing $S^{\mathrm{n}}(a_3)$ from the database we make the query false, in other words the tuple $R^{\mathrm{n}}(a_3, a_3)$ makes no difference, under any contingency. Notice that $\{R^{\mathrm{x}}(a_4, a_3)\}$ is not contingency because $R^{\mathrm{x}}(a_4, a_3)$ is exogenous.

In this paper we discuss two instantiations of query causality. In the first, called Why-So causality, we are given an actual answer $\bar{a}$ to the query, and would like to find the cause(s) for this answer. Definition 2.1 is given for Why-So causality. In this case D is the real database, and the endogenous tuples D^{n} are a given subset, while exogenous are $D^{\mathrm{x}} = D - D^{\mathrm{n}}$. In the second instantiation, called Why-No causality, we are given a non-answer $\bar{a}$ to the query, i.e. would like to know the cause why $\bar{a}$ is not an answer. This requires some minor changes to Definition 2.1. Now the real database consists entirely of exogenous tuples, D^{x}. In addition, we are given a set of potentially missing tuples, whose absence from the database caused $\bar{a}$ to be a non-answer: these form the endogenous tuples, D^{n}, and we denote $D = D^{\mathrm{x}} \cup D^{\mathrm{n}}$. We do not discuss in this paper how to compute D^{n}: this has been addressed in recent work [15]. In this setting, the definition of the Why-No causality is the dual of Def. 2.1 and we give it here briefly: a *counterfactual cause* for the non-answer $\bar{a}$ in D^{x} is a tuple $t \in D^{\mathrm{n}}$ s.t. $D^{\mathrm{x}} \not\models q(\bar{a})$ and $D^{\mathrm{x}} \cup \{t\} \models q(\bar{a})$; an actual cause for the non-answer $\bar{a}$ is a tuple $t \in D^{\mathrm{n}}$ s.t. there exists a set $\Gamma \subseteq D^{\mathrm{n}}$ called contingency set s.t. t is a counterfactual cause for the non-answer of $\bar{a}$ in $D^{\mathrm{x}} \cup \Gamma$.

We now define *responsibility*, measuring the degree of causality.

DEFINITION 2.3 (RESPONSIBILITY). *Let $\bar{a}$ be an answer or non-answer to a query q, and let t be a cause (either Why-So, or Why-No cause). The* responsibility *of t for the (non-)answer $\bar{a}$ is:*

$$\rho_t = \frac{1}{1 + \min_\Gamma |\Gamma|}$$

where Γ ranges over all contingency sets for t.

Thus, the responsibility is a function of the minimal number of tuples that we need to remove from the real database D (in the case of Why-So), or that we need to add to the real database D^{x} (in the case of Why-No) before it becomes counterfactual. The tuple t is a counterfactual cause iff $\rho_t = 1$, and it is an actual cause iff $\rho_t > 0$. By convention, if t is not a cause, $\rho_t = 0$.

EXAMPLE 2.4 (IMDB CONTINUED). *Figure 2a shows the lineage of the answer Musical in Example 1.1. Consider the movie "Sweeney Todd": its responsibility is* $1/3$ *because the smallest contingency is:* {`Director`*(David, Burton),* `Director`*(Humphrey, Burton)*} *(if we remove both directors, then "Sweeney Todd" becomes counterfactual). Consider now the movie "Manon Lescaut": its responsibility is* $1/5$ *because the smallest contingency set is* {`Director` *(David, Burton),* `Movie`*("Flight"),* `Movie`*("Candide"),* `Director`*(Tim, Burton)*}.

We now define formally the problems studied in this paper. Let $D = D^{\mathrm{x}} \cup D^{\mathrm{n}}$ be a database consisting of endogenous and exogenous tuples, q be a query, and $\bar{a}$ be a potential answer to the query.

Causality problem Compute the set $C \subseteq D^{\mathrm{n}}$ of actual causes for the answer $\bar{a}$.

Responsibility problem For each actual cause $t \in C$, compute its responsibility ρ_t.

We study the *data complexity* in this paper: the query q is fixed, and the complexity is a function of the size of the database instance D. In the rest of the the paper we restrict our discussion w.l.o.g. to Boolean queries: if $q(\bar{x})$ is not Boolean, then to compute the causes or responsibilities for an answer $\bar{a}$ it suffices to compute the causes or responsibilities of the Boolean query $q[\bar{a}/\bar{x}]$, where all head variables are substituted with the constants in $\bar{a}$.

3. COMPLEXITY OF CAUSALITY

We start by proving that causality can be computed efficiently; even stronger, we show that causes can be computed by a relational query. This is in contrast with the general causality problem, where Eiter [9] has shown that deciding causality for a Boolean expression is NP-complete. We obtain tractability by restricting our queries to conjunctive queries. Chockler et al. [6] have shown that causality for "read once" Boolean circuits is in PTIME. Our results are strictly stronger: for the case of conjunctive queries without self-joins, queries with read-once lineage expressions are precisely the hierarchical queries [8, 21], while our results apply to all conjunctive queries. The results in this section apply uniformly to both Why-So and Why-No causality, so we will simply refer to causality without specifying which kind. Also, we restrict our discussion to Boolean queries only.

We write positive Boolean expressions in DNF, like $\Phi = (X_1 \wedge X_3) \vee (X_1 \wedge X_2 \wedge X_3) \vee (X_1 \wedge X_4)$; sometimes we drop $\wedge$, and write $\Phi = X_1X_3 \vee X_1X_2X_3 \vee X_1X_4$. A conjunct c is *redundant* if there exists another conjunct c' that is a strict subset of c. Redundant conjuncts can be removed without affecting the Boolean expression. In our example, $X_1X_2X_3$ is redundant, because it strictly contains X_1X_3; it can be removed and Φ simplifies to $X_1X_3 \vee X_1X_4$. A positive DNF is *satisfiable* if it has at least one conjunct; otherwise it is equivalent to `false` and we call it *unsatisfiable*.

Next, we review the definition of lineage. Fix a Boolean conjunctive query consisting of m atoms, $q = g_1, \ldots, g_m$, and database instance D; recall that $D = D^{\mathrm{x}} \cup D^{\mathrm{n}}$ (exogenous and endogenous tuples). For every tuple $t \in D$, let X_t denote a distinct Boolean variable associated to that tuple. A *valuation* for q is a mapping, $\theta : Var(q) \rightarrow Adom(D)$, where $Adom(D)$ the active domain of the database, such that the instantiation of every atom is a tuple in the database: $t_i = \theta(g_i) \in D$ for $i = 1, \ldots, m$. We associate to the valuation θ the following conjunct: $c^\theta = X_{t_1} \wedge \ldots \wedge X_{t_m}$. The *lineage* of q is:

$$\Phi = \bigvee_{\theta : q \rightarrow D} c^\theta$$

We will assume w.l.o.g. that $D^{\mathrm{x}} \not\models q$ and $(D^{\mathrm{x}} \cup D^{\mathrm{n}}) \models q$ (otherwise we have no causes).

DEFINITION 3.1 (n-LINEAGE). *The* n-lineage *of q is:*

$$\Phi^{\mathrm{n}} = \Phi[X_t := \mathtt{true}, \forall t \in D^{\mathrm{x}}]$$

Here $\Phi[X_t := \mathtt{true}, \forall t \in D^{\mathrm{x}}]$ means substituting X_t with `true`, for all Boolean variables X_t corresponding to exogenous tuples t. Thus, the n -lineage is obtained as follows. Compute the standard lineage, over all tuples (exogenous and endogenous), then set to `true` all exogenous tuples: the remaining expression depends only on endogenous tuples. The following technical result allows us to compute the causes to answers of conjunctive queries.

THEOREM 3.2 (CAUSALITY). *Let q be a conjunctive query, and t be an endogenous tuple. Then the following three conditions are equivalent:*

1. *t is an actual cause for q (Def. 2.1).*
2. *There exists set of tuples $\Gamma \subseteq D^{\mathrm{n}}$ such that the lineage $\Phi[X_u = \mathtt{false}, \forall u \in \Gamma]$ is satisfiable, and $\Phi[X_u = \mathtt{false}, \forall u \in \Gamma; X_t = \mathtt{false}]$ is unsatisfiable.*
3. *There exists a non-redundant conjunct in the* n*-lineage Φ^{n} that contains the variable X_t.*

We give the proof in the Appendix. The theorem gives a PTIME algorithm for computing all causes of q: compute the n-lineage Φ^{n} as described above, and remove all redundant conjuncts. All tuples that still occur in the lineage are actual causes of q.

EXAMPLE 3.3. *Consider $q :- R(x, y), S(y), y = a_3$ over the database of Example 2.2. Its lineage is $\Phi = X_{R(a_3,a_3)}X_{S(a_3)} \vee X_{R(a_4,a_3)}\, X_{S(a_3)}$. Assume $R(a_4, a_3)$ is exogenous and $R(a_3, a_3)$, $S(a_3)$ are endogenous. Then the* n*-lineage Φ^{n} is obtained by setting $X_{R(a_4,a_3)} = \mathtt{true}$: $\Phi^{\mathrm{n}} = X_{R(a_3,a_3)}X_{S(a_3)} \vee X_{S(a_3)}$. After removing the redundant conjunct, the* n*-lineage becomes $\Phi^{\mathrm{n}} = X_{S(a_3)}$; hence, $S(a_3)$ is the only actual cause for the query.*

In the rest of this section we prove a stronger result. Denote C_R the set of actual causes in the relation R; that is, $C_R \subseteq R^{\mathrm{n}}$, and every tuple $t \in C_R$ is an actual cause. We show that C_R can be computed by a relational query. In particular, this means that the causes to a (non-)answer can be computed by a SQL query, and therefore can be performed entirely in the database system.

THEOREM 3.4 (CAUSALITY FO). *Given a Boolean query q over relations $R_1, \ldots R_k$, the set of all causes of q $\{C_{R_1}, \ldots, C_{R_k}\}$ can be expressed in non-recursive stratified Datalog with negation, with only two strata.*

Theorem 3.4, shows that causes can be expressed in a language equivalent to a subset of first order logic [1] and that, moreover, only one level of negation is needed. The proof is in the appendix.

EXAMPLE 3.5. *Continuing with the query $q :- R(x, y), S(y)$ from Example 3.3), suppose all tuples in S are endogenous. Thus, we have $R^{\mathrm{x}}, R^{\mathrm{n}}, S^{\mathrm{n}}$, but $S^{\mathrm{x}} = \emptyset$. The complete Datalog program that produces the causes for q is:*

$$\begin{aligned} I(y) &:- R^{\mathrm{x}}(x, y), S^{\mathrm{n}}(y) \\ C_R(x, y) &:- R^{\mathrm{n}}(x, y), S^{\mathrm{n}}(y), \neg I(y) \\ C_S(y) &:- R^{\mathrm{n}}(x, y), S^{\mathrm{n}}(y), \neg I(y) \\ C_S(y) &:- R^{\mathrm{x}}(x, y), S^{\mathrm{n}}(y) \end{aligned}$$

The role of $\neg I(y)$ is to remove redundant terms from the lineage. To see this, consider the database $R = \{(a_4, a_3), (a_3, a_3)\}$, $S = S^{\mathrm{n}} = \{a_3\}$, and assume that $R^{\mathrm{n}} = \{(a_3, a_3)\}$, $R^{\mathrm{x}} = \{(a_4, a_3)\}$, thus, q's lineage and n*-lineage are:*

$$\Phi = X_{R(a_4,a_3)} X_{S(a_3)} \vee X_{R(a_3,a_3)} X_{S(a_3)}$$
$$\Phi^{\mathrm{n}} = X_{S(a_3)} \vee X_{R(a_3,a_3)} X_{S(a_3)} \equiv X_{S(a_3)}$$

Thus, the only actual cause of q is $S(a_3)$. Consider C_R, which computes causes in R. Without the negated term $\neg I(y)$, C_R would return $R(a_3, a_3)$ (which would be incorrect). The role of the negated term $\neg I(y)$ is to remove the redundant terms in Φ^{n}: in our example, C_R returns the empty set (which is correct). Similarly, one can check that C_S returns $S(a_3)$. Note that negation is necessary in C_R because it is non-monotone: if we remove the tuple $R(a_4, a_3)$ from the database then $R(a_3, a_3)$ becomes a cause for the query q, thus C_R is non-monotone. Hence, in general, we must use negation in order to compute causes.

EXAMPLE 3.6. *Consider $q :- S(x), R(x, y), S(y)$, and assume that S is endogenous and R is exogenous: in other words, $S = S^{\mathrm{n}}$, $R = R^{\mathrm{x}}$. The following Datalog program computes all causes:*

$$I(x) :- S^{\mathrm{n}}(x), R^{\mathrm{x}}(x, x)$$
$$C_S(x) :- S^{\mathrm{n}}(x), R^{\mathrm{x}}(x, y), S^{\mathrm{n}}(y), \neg I(x), \neg I(y)$$
$$C_S(y) :- S^{\mathrm{n}}(x), R^{\mathrm{x}}(x, y), S^{\mathrm{n}}(y), \neg I(x), \neg I(y)$$

Here, too, we can prove that C_S is non-monotone and, hence, must use negation. Consider the database instance $R = \{(a_4, a_3), (a_3, a_3)\}$, $S = \{a_3, a_4\}$. Then $S(a_4)$ is not a cause; but if we remove $R(a_3, a_3)$, then $S(a_4)$ becomes a cause.

As the previous examples show, the causality query C is, in general, a non-monotone query: by inserting more tuples in the database, we determine some tuples to no longer be causes. Thus, negation is necessary in order to express C. The following corollary gives a sufficient condition for the causality query to simplify to a conjunctive query.

COROLLARY 3.7. *Suppose that each relation R_i is either endogenous or exogenous (that is, either $R_i^{\mathrm{n}} = R_i$ or $R_i^{\mathrm{x}} = R_i$). Further, suppose that, if R_i is endogenous, then the relation symbol R_i occurs at most once in the query q. Then, for each relation name R_i, the causal query C_{R_i} is a single conjunctive query (in particular it has no negation).*

The two examples above show that the corollary is tight: Example 3.5 shows that causality is non-monotone when a relation is mixed endogenous/exogenous, and Example 3.6 shows that causality is non-monotone when the query has self-joins, even if all relations are either endogenous or exogenous.

To illustrate the corollary, we revisit Example 3.5, where the query is $q :- R(x, y), S(y)$, and assume that $R^{\mathrm{x}} = \emptyset$ and $S^{\mathrm{x}} = \emptyset$. Then the Datalog program becomes:

$$C_R(x, y) :- R^{\mathrm{n}}(x, y), S^{\mathrm{n}}(y)$$
$$C_S(y) :- R^{\mathrm{n}}(x, y), S^{\mathrm{n}}(y)$$

4. COMPLEXITY OF RESPONSIBILITY

In this section, we study the complexity of computing responsibility. As before, we restrict our discussion to Boolean queries. Thus, given a Boolean query q and an endogenous tuple t, compute its responsibility ρ_t (Def. 2.3). We say that the query is in PTIME if there exists a PTIME algorithm that, given a database D and a tuple t computes the value ρ_t; we say that the query is NP-hard, or simply hard, if the problem "given a database instance D and a number v, check whether $\rho_t > v$" is NP-hard. The strongest result in this section and the paper is a *dichotomy theorem* for Why-So queries without self-joins: for every query, computing the *responsibility is either in PTIME or NP-hard* (Sect. 4.1). The case of non-answers (Why-No) turns out to be a simpler problem as Sect. 4.2 shows.

4.1 Why So?

We assume that the conjunctive query q is without self-joins, i.e. every relation occurs at most once in q; we discuss self-joins briefly at the end of the section. W.l.o.g. we further assume that each relation is either fully endogenous or exogenous ($R_i^{\mathrm{n}} = R_i$ or $R_i^{\mathrm{x}} = R_i$). Recall that computing the Why-So responsibility of a tuple t requires computing the smallest contingency set Γ, such that t is a counterfactual cause in $D - \Gamma$. We start by giving three hard queries, which play an important role in the dichotomy result.

THEOREM 4.1 (CANONICAL HARD QUERIES). *Each of the following three queries is NP-hard:*

$$h_1^* :- A^{\mathrm{n}}(x), B^{\mathrm{n}}(y), C^{\mathrm{n}}(z), W(x, y, z)$$
$$h_2^* :- R^{\mathrm{n}}(x, y), S^{\mathrm{n}}(y, z), T^{\mathrm{n}}(z, x)$$
$$h_3^* :- A^{\mathrm{n}}(x), B^{\mathrm{n}}(y), C^{\mathrm{n}}(z), R(x, y), S(y, z), T(z, x)$$

If the type of a relation is not specified, then the query remains hard whether the relation is endogenous or exogenous.

We give the proof in the Appendix: we prove the hardness of h_1^* and h_2^* directly, and that of h_3^* by using a particular reduction from h_2^*. Chockler and Halpern [5] have already shown that computing responsibility for Boolean circuits is hard, in general. One may interpret our theorem as strengthening that result by providing three specific queries whose responsibility is hard. However, the theorem is much more significant. We show in this section that every query that is hard can be proven to be hard by a simple reduction from one of these three queries.

Next, we illustrate PTIME queries, and start with a trivial example $q :- R(a, y)$ where a is a constant. If $t = R(a, b)$, then its minimum contingency is simply the set of all tuples $R(a, c)$ with $c \neq b$, and one can compute t's responsibility by simply counting these tuples. Thus, q is in PTIME. We next give a much more subtle example.

EXAMPLE 4.2 (PTIME QUERY). *Let $q :- R(x, y), S(y, z)$, let both R and S be endogenous, and w.l.o.g. let t be a tuple in R. We show how to compute the size of the minimal contingency set Γ for t with a reduction to the max-flow/min-cut problem in a network. Given the database instance D, construct the network illustrated in Fig. 4. Its vertices are partitioned into $V_1 \cup \ldots \cup V_5$. V_1 contains the* source*, which is connected to all nodes in V_2. There is one edge (x, y) from V_2 to V_3 for every tuple $(x, y) \in R$, and one edge (y, z) from V_3 to V_4 for every tuple $(y, z) \in S$. Finally, every node in V_4 is connected to the* target*, in V_5. Set the capacity of all edges from the source or into the target to ∞. The other capacities will be described shortly. Recall that a cut in a network is a set of edges Γ that disconnect the source from the target. A* min-cut *is a cut of minimum capacity, and the capacity of a min-cut can be computed in PTIME using Ford-Fulkerson's algorithm. Now we make an important observation: any mincut Γ in the network corresponds to a set of tuples[2] in the database $D = R \cup S$, such that q is false on $D - \Gamma$. We use this fact to compute the responsibility of t as follows: First, set the capacity of t to 0, and that of all other tuples in R, S to 1. Then, repeat the following procedure for every path p*

[2]In other words, the mincut cannot include the extra edges connected to the source or the target as they have infinite capacity.

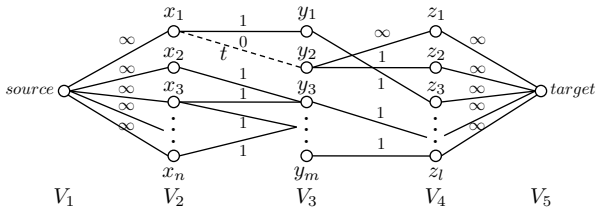
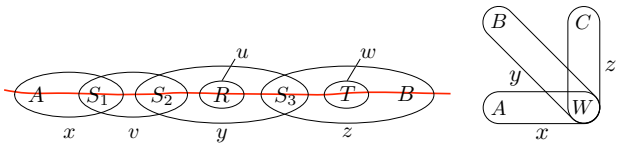


Figure 4: Flow transformation for $q :- R(x, y), S(y, z)$.



(a) q (b) h_1^*

Figure 5: Dual query hypergraphs for easy query $q :- A(x), S_1(x, v), S_2(v, y), R(y, u), S_3(y, z), T(z, w), B(z)$, and hard query $h_1^* :- A(x), B(y), C(z), W(x, y, z)$

from the source to the target that goes through t: set the capacities of all edges[3] *in $p - \{t\}$ to ∞, compute the size of the mincut, and reset their capacities back to 1. In Fig. 4 there are two such paths p: the first is x_1, y_2, z_1 (the figure shows the capacities set for this path), the other path is x_1, y_2, z_2. We claim that for every mincut Γ, the set $\Gamma - \{t\}$ is a contingency set for t. Indeed, q is false on $D - \Gamma$ because the source is disconnected from the target, and q is true on $(D - \Gamma) \cup \{t\}$, because once we add t back, it will join with the other edges in $p - \{t\}$. Note that Γ cannot include these edges as their capacity is ∞. Thus, by repeating for all paths p (which are at most $|S|$), we can compute the size of the minimal contingency set as* $\min |\Gamma| - 1$.

We next generalize the algorithm in Example 4.2 to the large class of *linear queries*. We need two definitions first.

DEFINITION 4.3 (DUAL QUERY HYPERGRAPH $\mathcal{H}^D(\mathcal{V}, \mathcal{E})$). *The* dual query hypergraph $\mathcal{H}^D(V, \mathcal{E})$ *of a query $q :- g_1, \ldots, g_m$ is a hypergraph with vertex set $V = \{g_1, \ldots, g_m\}$ and a hyperedge E_i for each variable $x_i \in Var(q)$ such that $E_i = \{g_j \mid x_i \in Var(g_j)\}$.*

Note that nodes are the atoms, and edges are the variables. This is the "dual" of the standard query hypergraph [11], where nodes are variables and edges are atoms.

DEFINITION 4.4 (LINEAR QUERY). *A hypergraph $H(V, \mathcal{E})$ is linear if there exists a total order S_V of V, such that every hyperedge $e \in \mathcal{E}$ is a consecutive subsequence of S_V. A query is* linear *if its dual hypergraph is linear.*

In other words a query is linear if its atoms can be ordered such that every variable appears in a continuous sequence of atoms. For example, the query q in Fig. 5a is linear. Order the atoms as $A, S_1, S_2, R, S_3, T, B$, and every variable appears in a continuous sequence, e.g. y occurs in S_2, R, S_3. On the other hand, none of the queries in Theorem 4.1 is linear. For example, the dual hypergraph of h_1^* is shown in Fig. 5b: one cannot "draw a line" through the vertices and stay inside hyperedges. Note that the definition of linearity ignores the endogenous/exogenous status of the atoms.

[3]In our example, $p - \{t\}$ contains a single other edge (namely a tuple in S). For longer queries, it may contain additional edges. For the query $R(x, y), S(y, z), T(z, u)$, for example, $p - \{t\}$ always contains two edges. Hence we refer to edges in $p - \{t\}$ in the plural.

Algorithm 1: Calculating responsibility for linear queries

Input: $q :- g_1, \ldots, g_m$, D and t, **Output**: ρ_t

$G =$ `flowGraph`$(dualHypergraph(q), D)$;
forall *source-target paths* $p = \{e_1, \ldots, e_m\} \in G$, $t \in p$ **do**
  `capacity`$(e_i) \leftarrow \infty$, `capacity`$(t) \leftarrow 0$;
  $\Gamma_j \leftarrow$ `maxFlow`(G) ;
return $\rho_t = \frac{1}{1+(min_j|\Gamma_j|-1)}$;

Function: `flowGraph`$(\mathcal{H}, D)$
$L = \{g_1(\bar{x}_1), \ldots, g_m(\bar{x}_m)\}$ linearization of $\mathcal{H}$;
$V = \{\{$`source`$\}, V_1', V_1, V_2', V_2 \ldots, V_m', V_m, \{$`target`$\}\}$;
forall g_i **do**
  $V_i' \leftarrow \{t_j \mid t_j \in q(\bar{x}_i') :- g_{i-1}, g_i\}$, $\bar{x}_i' \leftarrow \bar{x}_{i-1} \cap \bar{x}_i$;
  $\forall u \in V_{i-1}, v \in V_i'$, add edge $e(u, v)$ if $u(\bar{x}_i') = v(\bar{x}_i')$;
  `capacity`$(e) \leftarrow \infty$;
  forall $t_j \in g_i$ **do**
    $V_i \leftarrow V_i' \cup \{t_j\}$;
    $\forall u \in V_i', v \in V_i$, add edge $e(u, v)$ if $u(\bar{x}_i') = v(\bar{x}_i')$;
    if $t_j \in D^{\text{n}}$ **then** `capacity`$(e) \leftarrow 1$ **else** `capacity`$(e) \leftarrow \infty$;
$\forall v \in V_m$, `capacity`$(e(v,$ `target`$)) \leftarrow \infty$;
$\forall v \in V_1'$, `capacity`$(e($`source`$, v)) \leftarrow \infty$;

For every linear query, the responsibility of a tuple can be computed in PTIME using Algorithm 1. The algorithm essentially extends the construction given Example 4.2 to arbitrary linear queries. Note that it treats endogenous relations differently than exogenous by assigning to them weight ∞. Thus, we have:

THEOREM 4.5 (LINEAR QUERIES). *For any linear query q and any endogenous tuple t, the responsibility of t for q can be computed in PTIME in the size of the database D.*

So far, Theorem 4.1 has described some hard queries, and Theorem 4.5 some PTIME queries. Neither class is complete, hence we do not yet have a dichotomy yet. To close the gap we need to work on both ends. We start by expanding the class of hard queries.

DEFINITION 4.6 (REWRITING $\rightsquigarrow$). *We define the following rewriting relation on conjunctive queries without self-joins: q rewrites to q', in notation $q \rightsquigarrow q'$, if q' can be obtained from q by applying one of the following three rules:*

- DELETE x ($q \rightsquigarrow q[\emptyset/x]$): *Here, $q[\emptyset/x]$ denotes the query obtained by removing the variable $x \in Var(q)$, and thus decreasing the arity of all atoms that contained x.*
- ADD y ($q \rightsquigarrow q[(x, y)/x]$): *Here, $q[(x, y)/x]$ denotes the query obtained by adding variable y to all atoms that contain variable x, and thus increasing their arity, provided there exists an atom in q that contains both variables x, y.*
- DELETE g ($q \rightsquigarrow q - \{g\}$): *Here, g denotes an atom and $q - \{g\}$ denotes the query q without the atom g, provided that g is exogenous, or there exists some other atom g_0 s.t. $Var(g_0) \subseteq Var(g)$.*

Denote $\overset{*}{\rightsquigarrow}$ the transitive and reflexive closure of $\rightsquigarrow$. We show that rewriting always reduces complexity:

LEMMA 4.7 (REWRITING). *If $q \rightsquigarrow q'$ and q' is NP-hard, then q is also NP-hard. In particular, q is NP-hard if $q \overset{*}{\rightsquigarrow} h_i$, where h_i is one of the three queries in Theorem 4.1.*

EXAMPLE 4.8 (REWRITING). *We illustrate how one can prove that the query $q :- R(x, y), S(y, z), T(z, u), K(u, x)$ is hard, by*

rewriting it to h_2:

$$\begin{aligned} q &\rightsquigarrow R(x,y), S(y,z), T(x,z,u), K(u,x) && \textit{(add } x\textit{)} \\ &\rightsquigarrow R(x,y), S(y,z), T(x,z,u), K(u,x,z) && \textit{(add } z\textit{)} \\ &\rightsquigarrow R(x,y), S(y,z), T(x,z,u) && \textit{(delete } K\textit{)} \\ &\rightsquigarrow R(x,y), S(y,z), T(x,z) && \textit{(delete } u\textit{)} \end{aligned}$$

With rewriting we expanded the class of hard queries. Next we expand the class of PTIME queries. As notation, we say that two atoms g_i, g_j of a conjunctive query are *neighbors* if they share a variable: $\texttt{Var}(g_i) \cap \texttt{Var}(g_j) \neq \emptyset$.

DEFINITION 4.9 (WEAKENING $\multimap$). *We define the following* weakening *relation on conjunctive queries without self-joins: q weakens to q', in notation $q \multimap q'$, if q' can be obtained from q by applying one of the following two rules:*

- DISSOCIATION *If g^{x} is an exogenous atom and v_i a variable occurring in some of its neighbors, then let q' be obtained by adding v_i to the variable set of g^{x} (this increases its arity).*
- DOMINATION *If g^{n} is an endogenous atom and there exists some other endogenous atom g_0^{n} s.t. $Var(g_0^{\mathrm{n}}) \subseteq Var(g^{\mathrm{n}})$, then let q' be obtained by making g exogenous, g^{x}.*

Intuitively, a minimum contingency never needs to contain tuples from a dominated relation, and thus the relation is effectively exogenous. Along the lines of Lemma 4.7, we show the following for weakening:

LEMMA 4.10 (WEAKENING). *If $q \multimap q'$ and q' is in PTIME, then q is also in PTIME.*

Thus, weakening allows us to expand the class of PTIME queries. We denote $\overset{*}{\multimap}$ the transitive and reflexive closure of $\multimap$. We say that a query q is *weakly linear* if there exists a weakening $q \overset{*}{\multimap} q'$ s.t. q' is linear. Obviously, every linear query is also weakly linear.

COROLLARY 4.11 (WEAKLY LINEAR QUERIES). *If q is weakly linear, then it is in PTIME.*

Lemma 4.10 is based on the simple observation that a weakening produces a query q' over a database instance D' that produces the same output tuples as query q on database instance D. Weakening only affects exogenous and dominated atoms, which are not part of minimum contingencies, and therefore responsibility remains unaffected. This also implies an algorithm for computing responsibility in the case of weakly linear queries: find a weakening of q that is linear and apply Algorithm 1.

EXAMPLE 4.12. *We illustrate the lemma with two examples. First, we show that $q :- R^{\mathrm{n}}(x,y), S^{\mathrm{x}}(y,z), T^{\mathrm{n}}(z,x)$ is in PTIME by weakening with a dissociation:*

$$q \multimap R^{\mathrm{n}}(x,y), S^{\mathrm{x}}(x,y,z), T^{\mathrm{n}}(z,x) \qquad \textit{(dissociation)}$$

The latter is linear. Query q should be contrasted with h_2^* in Theorem 4.1: the only difference is that here S^{x} is exogenous, and this causes q to be in PTIME while h_2^* is NP-hard. Second, consider $q :- R^{\mathrm{n}}(x,y), S^{\mathrm{n}}(y,z), T^{\mathrm{n}}(z,x), V^{\mathrm{n}}(x)$. Here we weaken with a domination followed by a dissociation:

$$\begin{aligned} q &\multimap R^{\mathrm{x}}(x,y), S^{\mathrm{n}}(y,z), T^{\mathrm{x}}(z,x), V^{\mathrm{n}}(x) && \textit{(domination)} \\ &\multimap R^{\mathrm{x}}(x,y,z), S^{\mathrm{n}}(y,z), T^{\mathrm{x}}(z,x,y), V^{\mathrm{n}}(x) && \textit{(dissociation)} \end{aligned}$$

The latter is linear with the linear order $S^{\mathrm{n}}, R^{\mathrm{x}}, T^{\mathrm{x}}, V^{\mathrm{n}}$.

We say that a query q is *final* if it is not weakly linear and for every rewriting $q \rightsquigarrow q'$, the rewritten query q' is weakly linear. For example, each of h_1^*, h_2^*, h_3^* in Theorem 4.1 is final: one can check that if we try to apply any rewriting to, say, h_1^* we obtain a linear query. We can now state our main technical result:

THEOREM 4.13 (FINAL QUERIES). *If q is final, then q is one of h_1^*, h_2^*, h_3^*.*

This is by far the hardest technical result in this paper. We give the proof in the appendix. Here, we show how to use it to prove the dichotomy result.

COROLLARY 4.14 (RESPONSIBILITY DICHOTOMY). *Let q be any conjunctive query without self joins. Then:*

- *If q is weakly linear then q is in PTIME.*
- *If q is not weakly linear then it is NP-hard.*

PROOF. If q is weakly linear then it is in PTIME by Corollary 4.11. Suppose q is not weakly linear. Consider any sequence of rewritings $q = q_0 \rightsquigarrow q_1 \rightsquigarrow q_2 \rightsquigarrow \ldots$ Any such sequence must terminate as any rewriting results in a simpler query. We rewrite as long as q_i is not weakly linear and stop at the last query q_k that is not weakly linear. That means that any further rewriting $q_k \rightsquigarrow q'$ results in a weakly linear query q'. In other words, q_k is a final query. By Theorem 4.13, q_k is one of h_1^*, h_2^*, h_3^*. Thus, we have proven $q \overset{*}{\rightsquigarrow} h_j$, for some $j = 1, 2, 3$. By Lemma 4.7, the query q is NP-hard. □

Extensions. We have shown in Sect. 3 that causality can be computed with a relational query. This raises the question: if the responsibility of a query q is in PTIME, can we somehow compute it in SQL? We answer this negatively:

THEOREM 4.15 (LOGSPACE). *Computing the Why-So responsibility of a tuple $t \in D^{\mathrm{n}}$ is hard for* LOGSPACE *for the following query: $q :- R^{\mathrm{n}}(x,u_1,y), S^{\mathrm{n}}(y,u_2,z), T^{\mathrm{n}}(z,u_3,w)$*

Finally, we add a brief discussion of queries with self-joins. Here we establish the following result:

PROPOSITION 4.16 (SELF-JOINS). *Computing the responsibility of a tuple t for $q :- R^{\mathrm{n}}(x), S^{\mathrm{x}}(x,y), R^{\mathrm{n}}(y)$ is NP-hard. The same holds if one replaces S^{x} with S^{n}.*

We include the proof in the appendix. Beyond this result, however, queries with self-joins are harder to analyze, and we do not yet have a full dichotomy. In particular, we leave open the complexity of the query $R^{\mathrm{n}}(x,y), R^{\mathrm{n}}(y,z)$.

4.2 Why No?

While the complexity of Why-So responsibility turned out to be quite difficult to analyze, the Why-No responsibility is much easier. This is because, for any query q with m subgoals and non-answer $\bar{a}$, any contingency set for a tuple t will have at most $m-1$ tuples. Since m is independent on the size of the database, we obtain the following:

THEOREM 4.17 (WHY-NO RESPONSIBILITY). *Given a query q over a database instance D and a non-answer $\bar{a}$, computing the responsibility of $t \in D^{\mathrm{n}}$ over D^{n} is in PTIME.*

5. RELATED WORK

Our work is mainly related to and unifies ideas from work on *causality*, *provenance*, and *query result explanations*.

Causality. Causality is an active research area mainly in logic and philosophy with its own dedicated workshops (e.g. [24]). The idea of *counterfactual causality* (if X had not occurred, Y would not have occurred) can be traced back to Hume [20], and the best known counterfactual analysis of causation in modern times is due to Lewis [16]. Halpern and Pearl [13] define a variation they call *actual causality* which relies upon a graph structure called a causal network, and adds the crucial concept of a *permissive contingency* before determining causality. Chockler and Halpern [5] define the degree of *responsibility* as a gradual way to assign causality. Our definitions of Why-So and Why-No causality and responsibility for conjunctive queries build upon the HP definition, but simplify it and do not require a causal network. A general overview of causality in a database context is given in [17], while [19] introduces functional causality as an improved, more robust version of the HP definition.

Provenance. Approaches for defining data provenance can be mainly classified into three categories: how-, why-, and where-provenance [2, 4, 7, 12]. We point to the close connection between *why-provenance* and Why-So causality: both definitions concern the same tuples if all tuples in a database are endogenous[4]. However, our work extends the notion of provenance by allowing users to partition the lineage tuples into endogenous and exogenous, and presenting a strategy for constructing a query to compute all causes[5]. In addition, we can rank tuples according to their individual responsibilities, and determine a gradual contribution with counterfactual tuples ranked first.

Missing query results. Very recent work focuses on the problem of explaining missing query answers, i.e. why a certain tuple is not in the result set? The work by Huang et al. [15] and the Artemis [14] system present provenance for potential answers by providing *tuple insertions* or modifications that would yield the missing tuples. This is equivalent to providing the set of endogenous tuples for Why-No causality. Alternatively, Chapman and Jagadish [3] focus on the *operator* in the query plan that eliminated a specific tuple, and Tran and Chan [23] suggest an approach to automatically generate a modified query whose result includes both the original query's results as well as the missing tuple.

Our definitions of Why-So and Why-No causality highlight the symmetry between the two types of provenance ("positive and negative provenance") [19]. Instead of considering them in separate manners, we show how to construct Datalog programs that compute all Why-So or Why-No tuple causes given a partitioning of tuples into endogenous and exogenous. Analogously, responsibility applies to both cases in a uniform manner.

6. CONCLUSIONS

In this paper, we introduce causality as a framework for explaining answers and non-answers in a database setting. We define two kinds of causality, Why-So for actual answers, and Why-No for non-answers, which are related to the provenance of answers and non-answers, respecitively. We demonstrate how to retrieve all causes for an answer or non-answer using a relational query. We give a comprehensive complexity analysis of computing causes and their responsibilities for conjunctive queries: whereas causality is shown to be always in PTIME, we present a dichotomy for responsibility within queries without self-joins.

[4]Note that why-provenance (also called minimal witness basis) defines a set of sets. To compare it with Why-So causality, we consider the union of tuples across those sets.

[5]Note that, in general, Why-So tuples are *not* identical to the subset of endogenous tuples in the why-provenance.

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[3] A. Chapman and H. V. Jagadish. Why not? In *SIGMOD*, 2009.

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

[5] H. Chockler and J. Y. Halpern. Responsibility and blame: A structural-model approach. *J. Artif. Intell. Res. (JAIR)*, 22:93–115, 2004.

[6] H. Chockler, J. Y. Halpern, and O. Kupferman. What causes a system to satisfy a specification? *ACM Trans. Comput. Log.*, 9(3), 2008.

[7] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[8] N. Dalvi and D. Suciu. Management of probabilistic data: Foundations and challenges. In *PODS*, pages 1–12, Beijing, China, 2007. (invited talk).

[9] T. Eiter and T. Lukasiewicz. Complexity results for structure-based causality. *Artif. Intell.*, 142(1):53–89, 2002. (Conference version in *IJCAI*, 2002).

[10] T. Eiter and T. Lukasiewicz. Causes and explanations in the structural-model approach: Tractable cases. *Artif. Intell.*, 170(6-7):542–580, 2006.

[11] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001.

[12] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.

[13] J. Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach. Part I: Causes. *Brit. J. Phil. Sci.*, 56:843–887, 2005. (Conference version in *UAI*, 2001).

[14] M. Herschel, M. A. Hernández, and W. C. Tan. Artemis: A system for analyzing missing answers. *PVLDB*, 2(2):1550–1553, 2009.

[15] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1):736–747, 2008.

[16] D. Lewis. Causation. *The Journal of Philosophy*, 70(17):556–567, 1973.

[17] A. Meliou, W. Gatterbauer, J. Halpern, C. Koch, K. F. Moore, and D. Suciu. Causality in databases. *IEEE Data Engineering Bulletin*, Sept. 2010.

[18] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *CoRR*, abs/1009.2021, 2010.

[19] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. Why so? or Why no? Functional causality for explaining query answers. In *MUD*, 2010. Full version: CoRR abs/0912.5340 (2009).

[20] P. Menzies. Counterfactual theories of causation. Stanford Encylopedia of Philosophy, 2008.

[21] D. Olteanu and J. Huang. Secondary-storage confidence computation for conjunctive queries with inequalities. In *SIGMOD*, 2009.

[22] P. Senellart and G. Gottlob. On the complexity of deriving schema mappings from database instances. *PODS*, 2008.

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

[24] International multidisciplinary workshop on causality. IRIT, Toulouse, June 2009.

APPENDIX

A. PROOFS CAUSALITY

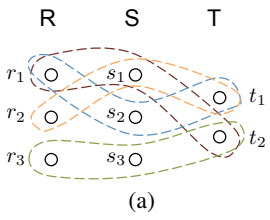
PROOF THEOREM 3.2. Assume $\Phi^{q(\bar{a})}$ the lineage of $\bar{a}$ in D. We construct the endogenous lineage $\Psi^{q(\bar{a})} = \Phi^{q(\bar{a})}(\bar{X}_{D^{\mathrm{x}}} = \mathtt{true})$, and Ψ' a DNF with all the non-redundant clauses of Ψ. We will show that a variable X_t is a cause of $\bar{a}$, iff $X_t \in \Psi'$, which means that X_t is part of a non-redundant clause in the endogenous lineage of $\bar{a}$.

Case A: (Why-So, answer $\bar{a}$): First of all, if X_t is not in Ψ', then X_t is not a cause of $\bar{a}$, as there is no assignment that makes X_t counterfactual for Ψ' (and therefore Ψ), because of monotonicity. If $X_t \in C_i$, where C_i a clause of Ψ', we select $\Gamma = \{X_j \mid X_j \in \Psi' \, and \, X_j \notin C_i\}$ ($\Gamma \subseteq D^{\mathrm{n}}$ since $\forall X_j \in \Psi'$, $X_j \in D^{\mathrm{n}}$). Then, if we write $\Psi' = C_i \vee \Psi''$, we know that $\Psi''(X_\Gamma = \mathtt{false}) = \mathtt{false}$, because Ψ' contains only non-redundant terms. That means that every clause C_j has at least one variable that is not in C_i, and therefore can be negated by the above choice of Γ. This makes X_t counterfactual for Ψ' (and also Ψ) with contingency Γ. Since $\Gamma \cap D^{\mathrm{x}} = \emptyset$ X_t is also counterfactual for $\Phi^{q(\bar{a})}$ with contingency Γ, meaning that $\Phi^{q(\bar{a})}(X_\Gamma = \mathtt{false})$ is satisfiable, and $\Phi^{q(\bar{a})}(X_\Gamma = \mathtt{false}, X_t = \mathtt{false})$ is unsatisfiable. Therefore, conditions 1, 2, and 3 are equivalent.

Case B: (Why-No, non-answer $\bar{a}$): First of all, if X_t is not in Ψ', then X_t is not a cause of $\bar{a}$, as there is no assignment that makes X_t counterfactual for Ψ' (and therefore Ψ), because of monotonicity. If $X_t \in C_i$, where C_i a clause of Ψ', we select $\Gamma' = \{X_j \mid X_j \in C_i \, and \, X_j \neq X_t\}$, and assign $\Gamma = D^{\mathrm{n}} - \Gamma' \cup \{t\}$. $\Gamma, \Gamma' \subseteq D^{\mathrm{n}}$ since $\forall X_j \in \Psi'$, $X_j \in D^{\mathrm{n}}$. Then, if we write $\Psi' = C_i \vee \Psi''$, we know that $\Psi''(X_\Gamma = \mathtt{false}) = \mathtt{false}$, because Ψ' contains only non-redundant terms. That means that every clause C_j has at least one variable that is not in C_i, and therefore can be negated by the above choice of Γ. This makes X_t counterfactual for Ψ' (and therefore Ψ) with contingency Γ'. Since $\Gamma' \cap D^{\mathrm{x}} = \emptyset$ X_t is also counterfactual for $\Phi^{q(\bar{a})}$ with contingency Γ', meaning that $\Phi^{q(\bar{a})}(X_\Gamma = \mathtt{false})$ is unsatisfiable, and $\Phi^{q(\bar{a})}(X_\Gamma = \mathtt{false}, X_t = \mathtt{false})$ is satisfiable. Therefore, conditions 1, 2, and 3 are equivalent. □

PROOF (THEOREM 3.4). To describe the relational query we need a number of technical definitions. Recall that $R_i^{\mathrm{n}}, R_i^{\mathrm{x}}$ denote the endogenous/exogenous tuples in R_i. Given a Boolean conjunctive query $q :- g_1(\bar{x}_1), \ldots, g_m(\bar{x}_m)$ we define a *refinement* to be a query of the form $r :- g_1^{\varepsilon_1}(\bar{x}_1), \ldots, g_m^{\varepsilon_m}(\bar{x}_m)$, where each $\varepsilon_i \in \{\mathrm{n}, \mathrm{x}\}$. Thus, every atom is made either exogenous or endogenous, and we call it an n- or an x-atom; there are 2^m refinements. Clearly, q is logically equivalent to the union of the 2^m refinements, and its lineage is equivalent to the disjunction of the lineages of all refinements. Consider any refinement r. We call a variable $x \in Var(r)$ an n-*variable* if it occurs in at least one n-atom. We apply repeatedly the following two operations: (1) choose two n-variables x, y and substitute $y := x$; (2) choose any n-variable x and any constant a occurring in the query and substitute $x := a$. We call any query s that can be obtained by applying these operations any number of times an *image* query; in particular, the refinement r itself is a trivial image. There are strictly less than 2^{k^2} images, where k is the total number of n-variables and constants in the query. Note that k is bounded by query size and thus irrelevant to data complexity. We always minimize an image query.

Fix a refinement r. We define an n-*embedding* for r as a function $e : r \to s$ that maps a strict subset of the n-atoms in r onto all n-atoms of s, where s is the image of a possibly different refinement r'. Intuitively, an n-embedding is a proof that a valuation for r results in a redundant conjunct, because it is strictly embedded in the conjunct of a valuation of r'.



(a)

R		S		T		W		
X		Y		Z		X	Y	Z
x_0		y_0		z_0		x_0	y_0	z_0
x_1		y_1		z_1		x_1	y_1	z_2
x_2		y_2		z_2		x_1	y_2	z_1
x_3		y_3				x_2	y_1	z_1
						x_3	y_3	z_2

(b)

Figure 6: (a) Example 3-partite 3-uniform hypergraph. (b) Database instance created from the hypergraph of (a).

We now describe in non-recursive, stratified Datalog the relational query that computes all causes

$$I_{s,e}(e(e^{-1}(\bar{y}))) :- \mathtt{atoms}(s) \quad (1)$$

$$C_{R_i}(\bar{x}_j) :- \mathtt{atoms}(r) \wedge \bigwedge_{e: r \to s} \neg I_s(e^{-1}(\bar{y})) \quad (2)$$

There is one IDB predicate $I_{s,e}(e(e^{-1}(\bar{y})))$, for each possible embedding $e : r \to s$, and it appears in one single rule, whose left hand side the same as s. The head variables are all n-variables $\bar{y}$ in s, where each y is repeated $|e^{-1}(y)|$ times: this is the purpose of the $e \circ e^{-1}$ function. For example, if the embedding is $e : R^{\mathrm{n}}(x_1, x_2, x_3) \to R^{\mathrm{n}}(y, y, y)$, then $e^{-1}(y) = (x_1, x_2, x_3)$ and $e(e^{-1}(y)) = (y, y, y)$, hence the IDB is $I(y, y, y)$. Next, there is one IDB predicate C_{R_i} for every relation name R_i, and there are one or more rules for C_{R_i}: one rule Eq. 2 for each refinement r and each n-atom $g_j^{\mathrm{n}}(\bar{x}_j)$ in r that refers to the relation R_i^{n}.

Therefore, the Datalog program consisting of Eq. 1 and Eq. 2 computes the set of all causes to the Boolean query q and returns them in the IDB predicates $C_1, \ldots, C_k$. □

PROOF (COROLLARY 3.7). The proof is immediate: there exists a single refinement, which has no embedding. □

B. CANONICAL HARD QUERIES

PROOF (THEOREM 4.1 $h_1^* :- R(x), S(y), T(z), W(x, y, z)$). We demonstrate hardness of q_1 with a reduction from the minimum vertex cover problem in a 3-partite 3-uniform hypergraph: Given an 3-partite, 3-uniform hypergraph and constant K, determine if there exists a vertex cover of size less or equal to K. This problem is shown to be hard in [22].

Take a 3-partite 3-uniform hypergraph such as the one from Fig. 6a. The nodes can be divided into 3 partitions (R, S and T), such that every edge contains exactly one node from each partition. We construct 4 database relations $R(x)$, $S(y)$, $T(z)$ and $W(x, y, z)$. For each node in the R partition of the hypergraph, we add a tuple in $R(x)$, and equivalently for S and T. Also, for each edge of the hypergraph, we add a tuple in $W(x, y, z)$. Finally, we add an additional tuple to each relation: $r_0 = (x_0)$, $s_0 = (y_0)$, $t_0 = (z_0)$ and $w_0 = (x_0, y_0, z_0)$.

The database instance corresponding to the hypergraph of Fig. 6a is shown in Fig. 6b. Now consider the join query

$$Q(x, y, z) :- R(x), S(y), T(z), W(x, y, z)$$

The responsibility of tuple r_0, (equivalently s_0, t_0 or w_0), is equal to $\frac{1}{1+|\mathcal{S}|}$, where $\mathcal{S}$ is the minimum contingency set for tuple r_0. Therefore, $\mathcal{S}$ contains the minimum number of tuples that make r_0 counterfactual. Note that s_0, t_0 and w_0 cannot be contained in $\mathcal{S}$,

as they are the only tuples that join with r_0. If S a minimum contingency, then if $w_i \in \mathcal{S}$, then $\exists \mathcal{S}' = \{\mathcal{S} \setminus \{w_i\}\} \cup \{r_j\}$, where $r_j.x = w_i.x$, and $\mathcal{S}'$ is also a contingency of the same size as S and therefore minimum. Therefore, there exists minimum contingency $\mathcal{S}$ that only contains tuples from relations R, S and T. The tuples of R, S and T correspond to hypergraph nodes, and a contingency corresponds to a cover: if an edge was not covered, then there would exist a corresponding tuple that was not eliminated. Also the cover is minimum: if there existed a smaller cover, then there would exist a smaller contingency. Therefore, computing responsibility for h_1^* is hard. □

LEMMA B.1. *Computing the minimum vertex cover in a* 3*-partite* 3*-uniform hypergraph* $H(V, \mathcal{E})$ *where* $\forall i \neq j \wedge E_i, E_j \in \mathcal{E} : |E_i \cap E_j| \leq 1$*, is NP-hard.*

PROOF LEMMA B.1. We will demonstrate this using a reduction from 3SAT. The proof is basically a modification of the 3-partite 3-uniform hypergraph vertex cover hardness result presented in [22]. Let $\phi \bigwedge_{i=1}^{n} c_i$ be an instance of 3SAT, where c_i are 3-clauses over some set X of variables. We build a 3-partite 3-uniform hypergraph (with vertex partitions $V = V_1 \cup V_2 \cup V_3$) as follows: for each variable $x \in X$ we add $12n$ nodes and $6n$ hyperedges to $\mathcal{H}$. $6n$ out of the $12n$ nodes are anonymous nodes appearing in only one hyperedge, and denoted by $\bullet$. Note the difference between this reduction and the one in [22]: [22] uses 12 nodes per variable, so for variable x it creates nodes $x_1, \bar{x}_1, x_2, \bar{x}_2, x_3, \bar{x}_3$, as well as 6 anonymous nodes. In our case, we replicate these to the number of all clauses n, so if we have 2 clauses, we would create the variables nodes $x_{11}, \bar{x}_{11}, x_{21}, \bar{x}_{21}, x_{31}, \bar{x}_{31}, x_{12}, \bar{x}_{12}, x_{22}, \bar{x}_{22}, x_{32}, \bar{x}_{32}$, and 12 anonymous nodes.

Note that $\forall j$, x_{ij} and $\bar{x}_{ij}$ belong to partition V_i ($i = \{1, 2, 3\}$). The subscript j in x_{ij} corresponds to clause j, therefore $j \in [1, n]$. We add local edges between same variable nodes according to the rules in Fig. 7.

V_1	V_2	V_3	
x_{1j}	$\bullet$	$\bar{x}_{3j'}$	$(j'=(j \mod n)+1)$
$\bullet$	x_{2j}	$\bar{x}_{3j}$	
$\bar{x}_{1j}$	x_{2j}	$\bullet$	
$\bar{x}_{1j}$	$\bullet$	x_{3j}	
$\bullet$	$\bar{x}_{2j}$	x_{3j}	
x_{1j}	$\bar{x}_{2j}$	$\bullet$	

Figure 7: Local hyperedges for variable x.

We add a global hyperedge for each clause c_j containing vertices that correspond to the variables in c_j, taking into account their position in the clause, and whether they are negated. For instance if $c_j = \bar{z} \vee x \vee y$, we add the hyperedge $(\bar{z}_{1j}, x_{2j}, y_{3j})$, which leaves the graph 3-partite. Our construction ensures that any two hyperedges will only have up to 1 node in common. An example of the reduction is given at Fig. 8 for the formula $(\bar{z} \vee x \vee y) \wedge (x \vee \bar{y} \vee z)$.

ϕ is satisfiable iff there is a vertex cover in $\mathcal{H}$ of size less than or equal to $3nm$, where m the cardinality of X and n the number of clauses. From this point the proof follows exactly the proof of Lemma 14 in [22] and is not repeated. □

PROOF (THEOREM 4.1 $h_2^* :- R(x, y), S(y, z), T(z, x)$). We show hardness with a reduction from the 3-partite 3-uniform hypergraph vertex cover where $\forall i \neq j \wedge E_i, E_j \in \mathcal{E} : |E_i \cap E_j| \leq 1$, which was shown to be NP-hard in Lemma B.1.

Assume a 3-partite 3-uniform hypergraph $\mathcal{H}(V, \mathcal{E})$ such that no two hyperedges share more than one node. Name the partitions V_R, V_S and V_T. For each node in V_R, V_S, V_T create a unique tuple in relations R, S and T respectively. We will create a graph

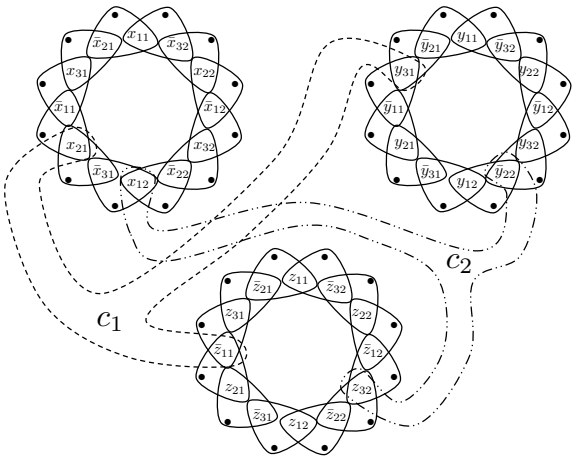


Figure 8: 3-partite 3-uniform graph transformation for 3SAT formula $(\bar{z} \vee x \vee y) \wedge (x \vee \bar{y} \vee z)$

G as follows: For each node in $r_i \in V_R$ create two nodes r_i^x and r_i^y. Similarly create nodes s_j^y, s_j^z for each $s_j \in V_S$, and t_k^x, t_k^z for each $t_k \in V_T$. For each edge $(r_i, s_j, t_k) \in \mathcal{E}$ create in G edges (r_i^y, s_j^y), (s_j^z, t_k^z) and (r_i^x, t_k^x). Each connected component in G contains nodes of equivalent variable types: x, y or z. Assign the same unique value across all the nodes of each connected component, which will reflect the assignment of the x, y and z variables of the R, S, T tuples. For every two tuples in the same relation, e.g $r_i(x_i, y_i)$ and $r_j(x_j, y_j)$, it is not possible that $x_i = x_j$ and $y_i = y_j$, because hyperedges of $\mathcal{H}$ do not have more than one node in common. That means that all created tuples are distinct, and each join result of q_2 corresponds to a hyperedge in $\mathcal{H}$. Finally, add tuples $R(x_0, y_0)$, $S(y_0, z_0)$ and $T(z_0, x_0)$, such that x_0, y_0 and z_0 do not match any x, y or z values respectively in the database. Computing the responsibility of tuple $R(x_0, y_0)$ corresponds to finding the minimum set of R, S, T tuples that remove all join results which is equivalent to a vertex cover in $\mathcal{H}$. Therefore, computing responsibility for query q_2 is NP-hard. □

PROOF (THEOREM 4.1 h_3^*). We prove hardness of h_3^* by a simple reduction from h_2^*. We start by writing the two queries as

$$h_2^* :- R(x, y), S(y, z), T(z, x)$$
$$h_3^* :- R'(x', y'), S'(y', z'), T'(z', x'), A'(x'), B'(y'), C'(z')$$

Now we transform a database instance for h_2^* into one of h_3^* as follows: For every tuple r_i in $R(x, y)$, insert r_i as new tuple into $A(x')$. Repeat analogously for each s_i, t_i from $S(y, z), T(z, x)$ and $B'(x'), C'(z')$. Then, for each valuation $\theta = [a/x, b/y, c/z]$ that makes h_2 true over D, insert (r_i, s_i), (s_i, t_i), and (t_i, r_i) into R', S', and T', respectively, where (r_i, s_i, t_i) represent the tuples in the original R, S, and T corresponding to θ. Now we have a one-to-one correspondence between a tuples in R and A', S and B', and T and C'. Comparing the lineages for these two queries, one sees that R', S', and T' are dominated by A', B', and C', and that the minimal lineages are identical. Hence causes and their responsibility are identical. □

C. RESPONSIBILITY DICHOTOMY

PROOF (THEOREM 4.5). It is straightforward from Algorithm 1. The flow graph constructed has one edge per database tuple. The capacities of exogenous tuples are ∞ and all other tuples have capacity one. Every unit of s-t flow corresponds to an output tuple of

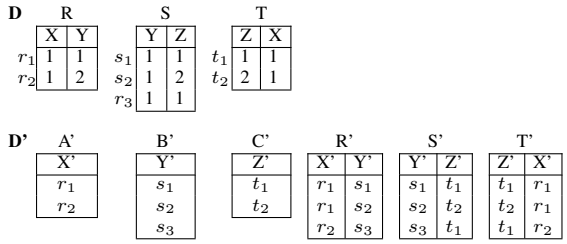


Figure 9: Example instance D for query h_2^* and corresponding instance D' for query h_3^*.

q. It is impossible that a flow does not correspond to a valid output tuple, as the partitions are ordered based on the linearization. This means that if a variable is chosen it will not occur again later in the flow, and therefore we can't have invalid flows going through one value of x in one partition and another in a later one. The steps of the algorithm: construction of the hypergraph, linearization, flow transformation, and computation of the maximum flow are all in PTIME and therefore responsibility of linear queries can be computed in PTIME. □

PROOF (LEMMA 4.7). Case 1: q' resulted from variable deletion ($q \rightsquigarrow q[\emptyset/x]$). Then we can polynomially reduce q' to q by setting variable x to the constant a. Therefore, if q is in PTIME, q' is also in PTIME.

Case 2: q' resulted from rewrite ($q \rightsquigarrow q[(x,y)/x]$). Assume $q :- g_1(x,\ldots)g_2(x,y,\ldots)q_0$, then $q' :- g_1'(x,y,\ldots)g_2'(x,y,\ldots)q_0'$. Reduce q' to q as follows: For each subgoal $g_1' \in q'$, create a unique value (x_i, y_j) for each tuple $g_1'(x_i, y_j)$, and assign these as the tuples of $g_1(x)$ in q. Similarly, create a unique value (x_i, y_j) for each tuple $g_2'(x_i, y_j)$, and assign each as a tuple $g_2((x_i, y_j), y_j)$ to form relation g_2. Both queries have the same output and the same contingencies. Therefore, if q is easy, q' also has to be easy.

Case 3: q' resulted from atom deletion ($q \rightsquigarrow q - \{g\}$). Assume $q :- g_1, \ldots, g_m$ and $q' :- g_1, \ldots g_{j-1}, g_{j+1}, \ldots, g_m$. q' differs from q in that it misses atom $g_j(\bar{y})$. The atom can be deleted with a rewrite only if it is exogenous, or $\exists g_i(\bar{x})$, with $\bar{x} \subseteq \bar{y}$.

We will reduce responsibility for q' to q: Take the output tuples of q' and project on $\bar{y}$. Assign the result to g_j. If g_j is exogenous or if $\bar{x} \subset \bar{y}$, then tuples from g_j are never picked in the minimum contingency. If $\bar{x} = \bar{y}$, no minimum contingency will have the same tuple from g_i and g_j. Therefore the contingency tuples from the 2 subgoals can be mapped to one of them, creating a contingency set for q' Therefore, if q is in PTIME, q' is also in PTIME. □

PROOF LEMMA 4.10. A weakening $q \multimap \hat{q}$ also results in a new database instance $\hat{D}$. A weakening does not create any new result tuples, and does not alter the number of tuples of any endogenous non-dominated atoms. Therefore contingencies in $\hat{q}, \hat{D}$ only contain tuples from non-weakened atoms making any contingency for q is also a contingency for $\hat{q}$ and vice versa. If any weakening results in a linear query $\hat{q}$, Algorithm 1 can be applied to solve it in PTIME, meaning that weakly linear queries are in PTIME. □

LEMMA C.1 (CONTAINMENT). *If q is final, then $\forall x, y$, $sg(x) \not\subseteq sg(y)$, where $sg(x)$ the set of subgoals of q that contain x.*

PROOF. Due to space restrictions, only a sketch of the proof is given here. For the full proof see [18].

We show the result by contradiction. Assume that $sg(x) \subseteq sg(y)$. The rewrites $q_1 :- q \rightsquigarrow q[\emptyset/x]$, $q_2 :- q \rightsquigarrow q[\emptyset/y]$, $q_3 :- q_2 \rightsquigarrow q_2[\emptyset/x]$, all have to be weakly linear. Also, because $sg(x) \subseteq sg(y)$, the connected components of q_3 are subsets of the connected components of q_1 and q_2. Using the linear orderings of the 3 rewritten queries we can produce a valid linear order for q, which would mean that q cannot be final, leading to a contradiction. □

LEMMA C.2. *If q is final, then it has exactly 3 variables.*

PROOF. Due to space restrictions, only a sketch of the proof is given here. For the full proof see [18].

First of all, if q has 2 or fewer variables, then it is linear, and therefore cannot be final. So, q must have at least 3 variables. Assuming that $Var(q) > 3$ leads to a contradiction. Since q is final, it cannot be linear. There are 2 cases: (i) $\forall g_i \in q$, $Var(g_i) \leq 2$, and (ii) $\exists g_i \in q$, $Var(g_i) \geq 3$. Case (i) is further subdivided into queries with corner points in their dual hypergraph, and those with cyclic dual hypergraphs. Since q is final, no rewrite can lead to a hard query. However, there exist rewrites that lead to one of h_1^*, h_2^*, h_3^*. Because of the assumption $Var(q) > 3$, it is in all cases guaranteed that at least one rewrite will occur, which means that q cannot be final.

Case (ii) is treated in the same manner. Using Lemma C.1 we identify 2 possible cases for q: (a) $A(x,y,\ldots), B(y,z,\ldots), C(z,x,\ldots)$, $z \notin A$, $x \notin B$ and $y \notin C$ and (b) $A(x,\ldots), B(y,\ldots), C(z,\ldots)$, $y, z \notin A$, $x, z \notin B$ and $x, y \notin C$, which can also be rewritten to hard queries and therefore cannot be final. □

PROOF (THEOREM 4.13). From Lemma C.2, since q is final, it has exactly 3 variables and it is not weakly linear. Let x, y, z be the 3 variables. An atom in q can be in one of the following 7 forms:

$$A(x),\ \ B(y),\ \ C(z),\ \ R(x,y),\ \ S(y,z),\ \ T(x,z),\ \ W(x,y,z)$$

Note that because of the third rewrite rule, and since q is final, we only need to consider queries with at most one atom of each type (e.g. a query containing $R_1(x,y)$ and $R_2(x,y)$ cannot be final). There are 7 possible atom types, and therefore 127 possibilities for queries. We do not need to analyze all of those, as most of them are trivially weakly linear (queries with less that 3 atoms, or less than 3 variables) and therefore cannot be final.

We will show that out of all the possible queries made up as combination of the 7 basic atoms, only h_1^*, h_2^*, h_3^* are final. The rest are either weakly linear or can be rewritten into the hard queries (and therefore are not final). Note that any query q whose atoms are a subset of the atoms of h_1^* or h_2^* is linear. Therefore, we only need to check supersets of h_1^* and h_2^*, and subsets of h_3^*. Any query where A, B, C are all endogenous, or R, S, T are all endogenous, is covered by these cases. We finally need to examine queries where at least one unary and one binary atom appear as exogenous. For simplicity, from now on we will drop the variable names and just use the relation symbols, in direct correspondence to the atom types mentioned above (e.g. we write A instead of $A(x)$ and R instead of $R(x,y)$). Also, if the endogenous or exogenous state is not explicit, it is assumed that the atom can be in either state.

Case 1: supersets of $h_1^* :- A^{\mathrm{n}}, B^{\mathrm{n}}, C^{\mathrm{n}}, W$. The only possible relations that can be added to h_1^* from the possible types are the binary relations R, S and T. For any of them, that are added to h_1^*, there exists a singleton relation (A, B or C) with a subset of their variables. Therefore, we can apply the third rewrite, eg. $q \rightsquigarrow q - \{R\}$ to get h_1^*. Therefore, any query q over variables x, y, z that is a superset of h_1^* is not final because it can be rewritten to h_1^*.

Case 2: supersets of $h_2^* :- R^{\mathrm{n}}, S^{\mathrm{n}}, T^{\mathrm{n}}$. The possible atom types that could be added are the ternary relation W, or the unary atoms A, B, and C. There are 5 possible cases excluding symmetries (adding A to h_2^* is equivalent to adding B instead). An atom in parentheses means that it may or may not be part of q.

(a) $q :- (A^{\mathrm{x}},)(B^{\mathrm{x}},)(C^{\mathrm{x}},)R^{\mathrm{n}}, S^{\mathrm{n}}, T^{\mathrm{n}}, W$: W (and $A^{\mathrm{x}}, B^{\mathrm{x}}, C^{\mathrm{x}}$) can be eliminated based on the third rewrite leading to h_2^*.

(b) $q :- A^{\mathrm{n}}, (B^{\mathrm{x}},)(C^{\mathrm{x}},)R, S, T(, W)$: weakly linear as A dominates R and T (and W), and B^{x} and C^{x} can be dissociated to W.

(c) $q :- A^{\mathrm{n}}, B^{\mathrm{n}}, (C^{\mathrm{x}},)R, S, T(, W)$: weakly linear as R, S, T (and W) are dominated, and C^{x} can be dissociated to W.

(d) $q :- A^{\mathrm{n}}, B^{\mathrm{n}}, C^{\mathrm{n}}, R, S, T$: this is h_3^*.

(e) $q :- A^{\mathrm{n}}, B^{\mathrm{n}}, C^{\mathrm{n}}, R, S, T, W$: W can be eliminated based on the third rewrite leading to h_3^*.

Case 3: subsets of $h_3^* :- A^{\mathrm{n}}, B^{\mathrm{n}}, C^{\mathrm{n}}, R, S, T$. We only need to consider those where at least one of the R, S, T atoms is exogenous or missing, otherwise they would fall under case 2. We have the following cases (excluding symmetries):

(a) $q :- A, B, C, R, S$: linear, and therefore any of its subsets are also linear.

(b) $q :- B^{\mathrm{n}}, C^{\mathrm{n}}, R, S, T^{\mathrm{x}}$: weakly linear as R, S, T are dominated and can dissociate to W.

(c) $q :- C^{\mathrm{n}}, R, S, T^{\mathrm{x}}$: T dissociates to W resulting in a linear query. Any subsets would also be linear.

Case 4: at least one exogenous unary and one exogenous binary relation.

(a) $q :- A^{\mathrm{x}}, B, C, R^{\mathrm{x}}, S, T, W$: A and R dissociate to W, resulting in a linear query. Any subset is also weakly linear.

(b) $q :- A^{\mathrm{x}}, B, C, R, S^{\mathrm{x}}, T, W$: A and S dissociate to W, resulting in a linear query. Any subset is also weakly linear.

Therefore, we have shown that any final query has to be one of h_1^*, h_2^*, h_3^*. □

D. OTHER RESPONSIBILITY PROOFS

PROOF (THEOREM 4.15). We will show the result through a series of reductions. We will start by a known LOGSPACE complete problem, the *Undirected Graph accessibility Problem* (UGAP): given an undirected graph $G = (V, E)$ and two nodes $a, b \in V$, decide whether there exists a path from a to b.

BGAP reduction: We define the *Bipartite Graph Accessibility Problem* (BGAP): given a bipartite graph (X, Y, E) and two nodes $a \in X, b \in Y$, decide whether there exists a path from a to b. Here the path is allowed to traverse edges in both directions, from X to Y and from Y to X. We will reduce any instance of UGAP to an instance of BGAP as follows:

Given an instance of UGAP as an undirected graph $G = (V, E)$, and nodes $a, b \in V$, construct a bipartite graph with $X = V$, $Y = E \cup \{c\}$, where c is a new node, and edges are of the form $(x, (x, y))$ and$(y, (x, y))$, plus one edge (b, c). Then there exists a path $a \to b$ in G iff there exists a path $a \to c$ in the bipartite graph. Therefore, BGAP is hard for LOGSPACE.

FPMF reduction: We define the *Four-Partite Max-Flow* problem (FPMF): given a four-partite network (U, X, Y, V, E) where each edge capacity is either 1 or 2, source and target nodes s and t connected to all nodes in U and V respectively with infinite capacities, and a number k, decide whether the max-flow is $\geq k$. We reduce BGAP to FPMF as follows:

Given an instance of BGAP as a bipartite graph (X, Y, E) and two nodes $a \in X, b \in Y$, construct a 4-partite graph (U, X, Y, V, E') by leaving the X and Y partitions and edges between them unchanged, as they are in the BGAP instance, and set their capacities to 2. Create a U-node xy for each edge $(x, y) \in E$. Each node $xy \in U$ is connected to $x \in X$ with an edge of capacity 1. Symmetrically, the V-nodes are E, and each node $y \in Y$ is connected to all nodes $xy \in V$, with capacity 1. Finally connect a source node s to all nodes U with infinite capacity, and connect all nodes in V to a target node t also with infinite capacity. The resulting graph has a maximum flow (min-cut) equal to $|E|$: the number of edges between any 2 partitions is exactly equal to E, and edges between the X and Y partitions are not chosen in a minimum cut, as they have capacity 2 instead of 1. The maximum flow of E utilizes all $U - X$ and $Y - V$ edges, and a residual flow of 1 is left in all $X - Y$ edges.

Now add to the graph a new node a' in partition U connected with capacity 1 to node a in X and with infinite capacity to the source node. Also add a node b' to partition V, connected to node b of partition Y with capacity 1, and to the target node with infinite capacity. The flow in this final graph is $|E|$ iff there is no path between a and b in the BGAP instance, and it is $|E| + 1$ iff there is a path between a and b. Therefore, BGAP can be solved by computing the maximum flow in the FPMF instance with $k = |E| + 1$.

Query reduction: We will reduce FPMF to computing responsibility for query q. Let (X, Y, Z, W, E) and number k be an instance of FPMF. For each (x_i, y_j) edge between partitions X and Y create a tuple $R(x_i, 1, y_j)$ is the capacity of the edge is 1, and two tuples $R(x_i, 1, y_j)$ and $R(x_i, 2, y_j)$ if the capacity of the edge is 2. Similarly create relation $S(y, u_2, z)$ based on the $Y-Z$ edges, and relation $T(z, u_3, w)$ based on the $Z - W$ edges. Finally, add tuples $R(x_0, 1, y_0)$, $S(y_0, 1, z_0)$, and $T(z_0, 1, w_0)$, where x_0, y_0, z_0 and w_0 are unique new values to the respective domains. The max-flow in the FPMF instance is $\geq k$ iff the responsibility of $R(x_0, 1, y_0)$ is $\geq k$. Therefore, responsibility for q is hard for LOGSPACE. □

PROOF (PROP. 4.16 SELF-JOINS). This results from a reduction from vertex cover. Given graph $G(V, E)$ as an instance of a vertex cover problem, we construct relations R and S as follows:

- For each vertex $v_i \in V$, create a new tuple r_i with unique value of attribute x_i.
- For each edge $(v_i, v_j) \in E$, create a new tuple s_k, with values $(x, y) = (x_i, x_j)$, where x_i, x_j are the values of tuples r_i, r_j that correspond to nodes v_i, and v_j respectively.
- Add tuples r_0 with value x_0 and s_0 with value (x_0, x_0).

The above transformation is polynomial, as we create one tuple per node and one tuple per edge. A vertex cover of size K in G is a contingency of size K for tuple r_0 in the database instance: removing from the database all tuples r_i corresponding to the cover leaves no other join result apart from the one due to r_0, s_0: All other $s_i = (x_i, y_i) \not\equiv s_0$ do not produce a join result, as at least one of $R(x_i)$ or $R(y_i)$ has been removed.

Now assume a contingency $\mathcal{S}$ for tuple r_0. If $\mathcal{S}$ contains a tuple $s_i = (x_i, y_i)$, then we can construct a new contingency $\mathcal{S}' = (\mathcal{S} \setminus \{s_i\}) \cup \{R(x_i)\}$, and $|\mathcal{S}'| \leq |\mathcal{S}|$. Therefore, there exists a minimum contingency $\mathcal{S}$ that contains only R-tuples. If V' the set of nodes that corresponds to tuples $r_i \in \mathcal{S}$, then V' is a vertex cover in G. If there was an edge left uncovered, then that means that there would be a tuple $S(x_i, y_i)$, such that neither of $R(x_i)$, $R(y_i)$ are in the contingency, which is a contradiction as the join tuple $R(x_i), S(x_i, y_i), R(y_i)$ would then be in the result. The cover V' is minimal, because $\mathcal{S}$ is minimal. □

PROOF (THEOREM 4.17 WHY-NO RESPONSIBILITY). This is a straightforward result based on the observation that the contingency set of a non-answer is bounded by the query size, and is therefore irrelevant to data complexity. In order to make a tuple counterfactual, we need to insert at most $m - 1$ tuples to the database, where m the number of query subgoals. □