Answering Why-Not Questions

Nicole Bidoit, Melanie Herschel, Katerina Tzompanaki

Answering Why-Not questions

Nicole Bidoit Melanie Herschel Katerina Tzompanaki
LRI - Université Paris Sud 11 / INRIA Saclay Île-de-France, 91405 Orsay Cedex, France
firstname.lastname@lri.fr

Abstract

Avec la masse croissante des données disponibles et la quantité des traitements ou transformations qui leur sont appliquées, il est devenu essentiel d'être capable de procéder à l'analyse en vue de la correction de ces traitements. Un des problèmes issus de la correction des transformations de données consiste à comprendre pourquoi certaines données ne figurent pas dans le résultat d'une requête relationnelle par exemple. Une possibilité pour expliquer cette absence de résultat, est d'identifier quelle partie de la requête est responsable de la perte de certaines données nécessaires à la production du résultat attendu mais non obtenu. Des premiers travaux (*why-not provenance*) ont été mené dans cette direction mais se sont avérés présenter des lacunes.

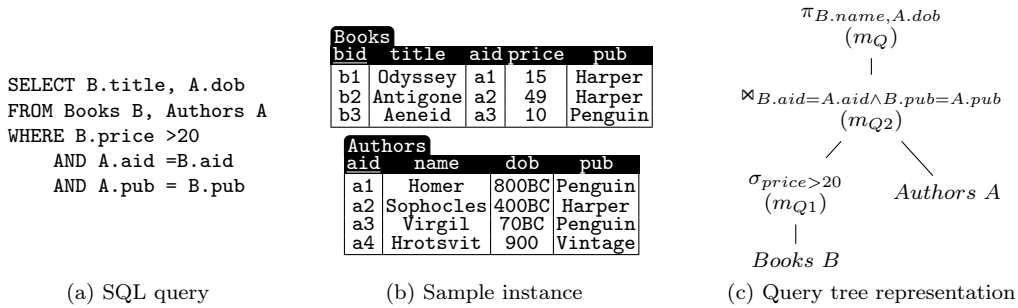
De manière à résoudre ces insuffisances, nous proposons un algorithme permettant d'extraire une explication aux données absentes du résultat d'une requête. Cet algorithme permet de calculer cette explication pour des requêtes relationnelles de selection, projection, jointure et union. Après l'introduction des concepts sur lesquels se base l'algorithme, celui-ci est décrit en détail. Une évaluation comparative montre que cet algorithme, d'une part, produit des explications plus pertinentes et d'autre part, est plus efficace relativement à l'état de l'art.

With the increasing amount of available data and transformations manipulating the data, it has become essential to analyze and debug data transformations. A sub-problem of data transformation analysis is to understand why some data are not part of the result of a relational query. One possibility to explain the lack of data in a query result is to identify where in the query data pertinent to the expected, but missing output is lost during query processing. A first approach to this so called why-not provenance *has been recently proposed, but we show that this first approach has some shortcomings.*

To overcome these shortcomings, we propose an algorithm to explain non-existing data in a query result. This algorithm allows to compute the why-not provenance for relational queries involving selection, projection, join and union. After providing necessary definitions, this paper contributes a detailed description of the algorithm. A comparative evaluation shows that it is both more efficient and effective than the state-of-the-art approach.

1 Introduction

In designing data transformations, for instance for data cleaning tasks, developers often face the problem that they cannot properly inspect or debug the individual steps of their transformation specification, which is commonly specified declaratively. All they see is the result data and, in case it does not correspond to their intent, developers have no choice but to manually analyze, fix, and test the data transformation again. For instance, a developer may wonder why some



(a) SQL query (b) Sample instance (c) Query tree representation

Figure 1: SQL query (a), instance (b), and query tree (c) of running example

products are missing from the result. Possible reasons for such *missing-answers* abound, e.g., were product tuples filtered by a particular selection or are join partners missing? Usually, a developer tests several modified versions of the original data transformation that are targeted towards identifying the reason for the missing tuples, for example by removing a selection predicate and observing if the products then appear in the result. Based on the result of this analysis, the query has to be fixed. For instance, the developer may decide to try a left outer join instead of a natural join. They then test the query again to see if their expectations are finally met and they also need to verify that their changes did not break anything else. Developers may undergo several such manual *analyze-fix-test (AFT)* cycles before reaching the expected result, a tedious and error-prone task.

Our research project *Nautilus* [1, 2] aims at semi-automatically supporting a developer in the AFT process by providing suited algorithms and tools. One important sub-problem during query analysis and debugging is the explanation of missing-answers. Further use-cases of finding missing-answers include what-if analysis focusing on the behavior of a query or the generation of queries for benchmarking purposes, where generated queries ideally do not return an empty result.

Very recently, approaches to explain missing-answers of relational and SQL queries have been proposed. In this paper, we focus on algorithms producing so called query-based explanations, illustrated by the following example.

Example 1.1. *Consider the SQL query shown in Fig. 1(a), both in its SQL and relational query tree form. Ignore the operator labels m_i in the query tree for now. The query finds expensive books and the period around which they were written. Such a query may for instance be used to determine if contemporary books are priced more than classical ones. Let us further assume the database instance* `Bib` *shown in Fig. 1(b), where* `bid` *is a primary key in* Books *and* `aid` *is a primary key in* Authors.

Now, let us assume that the tuple $t =$ (`Odyssey, 800BC`) is not included in the result of the query, although the developer or an analyst expected it to appear in the result as they are sure that these values exist in the database. Hence, for this why not question *(the missing tuple), we can find two query-based explanations in the form of* picky *subqueries: (1) the selection on price (indeed, the source tuple (`b1, Odyssey, a1, 15`), which is a candidate for contributing the value `Odyssey` to t has a price below 20, i.e., the output of the selection has no* compatible successor *for the source tuple) and/or (2) the join on publisher (indeed the publisher of the author with `dob = 800BC`, is not the same as the publisher of the book `Odyssey`).*

As we will discuss in detail, in using the state-of-the art algorithm for why-not provenance [3], we will only obtain the second solution, and we argue that this is one shortcoming of this approach as it does not provide the complete picture. Intuitively, when debugging the

query, removing the join on publisher attributes from the two relations will not solve the problem of the missing tuple. Due to this and other limitations of the existing algorithm, called *Why-Not algorithm* in the sequel, we propose a novel algorithm, named *NedExplain* [1]. More specifically, the scientific contributions of this paper are:

Detailed analysis of Why-Not. We review the state-of-the art algorithm Why-Not [3] and show that it has the following shortcomings: (1) queries involving self-joins are incorrectly processed, i.e., a wrong result, or no result is returned although there exists a query-based explanation; (2) Why-Not returns only partial results that depend on the logical tree representation of the query; and (3) Why-Not may return unexpected results because of its adapted (i) compatible and (ii) successor finding procedures.

Extended definition of query-based explanations. We provide, for the first time, a formal definition of query-based explanations. This definition goes beyond the concepts introduced previously, as it covers the special cases that are not well treated.

The NedExplain Algorithm. We propose NedExplain, an algorithm that correctly computes query-based explanations given a monotone relational query (involving selection, projection, join, and union operators) and a specification of a missing answer, within the framework provided by our definitions. An experimental study shows that NedExplain overall produces more satisfactory results than Why-Not.

The remainder of this paper is structured as follows. We set the theoretical foundation of our algorithm by providing necessary definitions in Sec. 2, before we discuss NedExplain in detail in Sec. 3. A comparative evaluation is presented in Sec. 4. We review related work in Sec. 5, with a special focus on previous work on generating query-based explanations. Finally, we conclude in Sec. 6.

2 Query-Based Explanation

In this section, we set the foundations for computing query-based explanations by providing necessary definitions. We assume that the reader is familiar with the relational model [4], and we only briefly revisit relevant notions in our context, in Sec. 2.1. We then formalize the why-not question to describe the data missing from a query result, in Sec. 2.2. In Sec. 2.3, we introduce the basic notions necessary to trace data throughout queries, before we more precisely define how we determine the culprit operators in Sec. 2.4. Finally, a formal definition of the why-not answer, i.e., the definition of our query-based why-not provenance, is given in Sec. 2.5.

2.1 Relational Preliminaries

Data model. A *tuple* t is a list of attribute-value pairs of the form $(A_1{:}v_1, \ldots, A_n{:}v_n)$. The *type* of a tuple t, denoted as $type(t)$, is the set of attributes occurring in t. For conciseness, we may omit attribute names when they are clear from the context, i.e., write $(v_1, \ldots, v_n)$.

A *relation schema* of a relation R is specified by $type(R)=\{R.A_1, \ldots, R.A_n\}$. Note that each attribute name A_i in $type(R)$ is qualified by the relation name R.

[1] The name is inspired by the name of one of the Nautilus' passengers in Jules Verne's novel 20,000 Leagues under the see, and also stands for non-existing-data-explain.

A *database instance* $\mathcal{I}$ over a database schema $\mathcal{S}=\{R_1, \ldots R_n\}$ is a mapping assigning to each R_i in $\mathcal{S}$, an instance $\mathcal{I}|_{R_i}$ over R_i. For the sake of presentation, we sometimes consider a database instance $\mathcal{I}$ as a set of tuples (of possibly different types).

Because relation schema attributes are qualified, two relation schemas will always have disjoint types. Thus, in order to define natural join and union, we need to introduce a renaming operation whose purpose is to assign the same attribute name to two attributes of distinct relations. We choose to enforce that renaming be based on new attribute names.

Definition 2.1 (Renaming ν)**.** *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two disjoint types. A renaming ν w.r.t. $\mathcal{T}_1$ and $\mathcal{T}_2$ is a set of triples (A_1, A_2, A_{new}) where $A_1 \in \mathcal{T}_1$, $A_2 \in \mathcal{T}_2$ and $A_{new} \notin \mathcal{T}_1 \cup \mathcal{T}_2$ is a new unqualified attribute. The co-domain of a renaming ν, denoted $cod(\nu)$ is the set $\{A_{new} \mid (A_1, A_2, A_{new}) \in \nu\}$.*

To further clarify this definition, consider the two types $type(Books)$ and $type(Authors)$ from our running example. A valid renaming with respect to these two types is: $(Books.aid, Authors.aid, authorID)$, which renames the two distinct attributes $Books.aid$ and $Authors.aid$ (coming from $type(Books)$ and $type(Authors)$ respectively), into a new attribute, named $authorID$.

Renamings are used especially in the context of queries and enable us to trace the (possible) origin of missing result tuples.

We now define a query Q. Essentially, we consider unions of conjunctive queries. Extending our methods to queries involving aggregation and set difference is part of future work.

Definition 2.2 (Query Q)**.** *Let $\mathcal{S}=\{R_1, \ldots, R_n\}$ be a database schema. Then*

1. *$[R_i]$ is a query with input schema R_i and target type $type(R_i)$, $i \in [1, n]$. $[R_i]$ has no proper sub-query.*

2. *Let Q_1, Q_2 be queries with input schemas $\mathcal{S}_1$, $\mathcal{S}_2$, target types $type(Q_1)$, $type(Q_2)$ and associated renamings (Q_1, ν_{Q_1}), (Q_2, ν_{Q_2}) respectively. Assuming that $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$, we have:*

   - *$[Q_1] \bowtie_\nu [Q_2]$ is a query Q where ν is a renaming w.r.t. $type(Q_1)$ and $type(Q_2)$. The input schema of Q is $\mathcal{S}_1 \cup \mathcal{S}_2$. Its target type $\nu(type(Q_1)) \cup \nu(type(Q_2))$.*
   - *$[Q_1] \cup_\nu [Q_2]$ is a query Q where ν is a renaming w.r.t. $type(Q_1)$ and $type(Q_2)$ if $\nu(type(Q_1)) = \nu(type(Q_2))$. The input schema of Q is $\mathcal{S}_1 \cup \mathcal{S}_2$ and its target type is $\nu(type(Q_1))$.*
   - *$\pi_W[Q_1]$ where $W \subseteq type(Q_1)$, is a query Q with input schema $\mathcal{S}_1$ and target type W.*
   - *$\sigma_C[Q_1]$ where C is a condition over $type(Q_1)$, is a query Q with input schema $\mathcal{S}_1$ and target type $type(Q_1)$.*

To simplify our discussion, we assume that every subquery of Q is named. Also, queries are defined such that two subqueries Q_1 and Q_2 have distinct target attributes. This query form is important for the development of our definitions and algorithms. It is of course clear that users may write their queries in a less restrictive way and these queries can be rewritten in the appropriate form afterwards. As final note, when dealing with complex queries, e.g., a query Q built from the queries Q_1 and Q_2, we consider as Q's subqueries both Q_1 and Q_2 as well as their respective subqueries.

We further define an input instance for a query Q whose input schema is $\mathcal{S}_Q$, as an instance over $\mathcal{S}_Q$. Finally, we will use a common tree representation of queries, where leaves correspond to relation instances and inner nodes to subqueries.

To correctly deal with queries involving self-joins, we introduce the following definition.

Definition 2.3 (Query over a database). *A query over a database schema $\mathcal{S}$ is given by a pair (Q, η_Q) where (1) Q is a query with input schema $\mathcal{S}_Q$ and (2) η_Q is a mapping from $\mathcal{S}_Q$ to $\mathcal{S}$ s.t.* for any $R \in \mathcal{S}$, $R.A \in type(R)$ iff $\eta_Q(R).A \in type(\eta_Q(R))$.

Assume a database instance $\mathcal{I}$ over $\mathcal{S}$. The evaluation of (Q, η_Q) over $\mathcal{I}$ is given by the evaluation of Q over the input instance $\mathcal{I}_Q$ over $\mathcal{S}_Q$ defined by:

$$\text{for any } S \in \mathcal{S}_Q, \quad \mathcal{I}_Q|_S = \mathcal{I}|_R \quad \text{if } \eta_Q(S) = R$$

The above notions set the stage for the formalization of query-based why-not provenance. We start by a formal definition of the why-not question one may ask.

2.2 The Why-Not Question

Intuitively, we specify the why-not question by means of a predicate characterizing the data which is missing from a query result. This predicate has the form of a disjunction of so called *conditional tuples*, which are essentially attribute-value pairs on which conjunctive predicates may be imposed. The following discussion introduces conditional tuples as well as predicates.

Definition 2.4 (v-tuple). *Let V be an enumerable set of variables. A v-tuple t_v of type $\{A_1, \ldots, A_n\}$ is of the form $(A_1{:}e_1, \ldots, A_n{:}e_n)$ where $e_i \in V \cup dom(A_i)$ for $i \in [1, n]$ and $dom(A_i)$ denoting the active domain of A_i.*

The variables of a v-tuple are similar in spirit to labeled nulls, which are used for instance in the context of data exchange [5]. Intuitively, the semantics associated to such variable is that we do not care about the value of the corresponding attribute.

In general, we want to be able to express that, although the actual value is unknown, the value should satisfy some constraints. For this reason, we define conditional tuples (or c-tuples for short) in a similar way as for incomplete databases [6].

Definition 2.5 (conditional tuple (c-tuple)). *Let t_v be a v-tuple and let X be the set of variables in t_v. A* c-tuple *t_c is a pair $(t_v, cond)$ where $cond = \bigwedge_{i=1}^{n} pred_i$ and for $1 \leq i \leq n$, $pred_i :: true \mid x_1\ cop\ x_2 \mid x_1\ cop\ a$, where x_i is a variable in X, $a \in dom(type(x_1))$, and cop is a comparison operator $(\neq, =, <, >, \geq, \leq)$.*

The type of a c-tuple $(t_v, cond)$ is of course the type of t_v.

Example 2.1. *Two possible conditional tuples are* $((\texttt{B.author:Sophocles}), (true))$ *and* $((\texttt{B.author:Sophocles,B.price:}x_1), (x_1 \neq 10 \ \wedge \ x_1 < 20))$.

We now define our Why-Not question based on a predicate. This predicate is a disjunction of conditional tuples and may include conditional tuples of different types, as defined next.

Definition 2.6 (Why-Not question). *A Why-Not question w.r.t. a query Q is a predicate $\mathcal{P}$ over Q's target type $\mathcal{T}_Q$, where $\mathcal{P} = \bigvee_{i=1}^{n} t_c^i$, t_c^i being a c-tuple s.t. $type(t_c^i) \subseteq \mathcal{T}_Q$.*

Example 2.2. *In the running example, we have intuitively expressed a Why-Not question as "Why is tuple $t =$ (`Odyssey, 800BC`) not in the result set?" To formally define the Why-Not question w.r.t. the query Q recall first that the output type of Q is $\mathcal{T}_Q = \{B.title, A.dob\}$. So, the Why-Not question is represented by predicate $\mathcal{P} = ((\texttt{B.title:Odyssey, A.dob:800BC}), (true))$.*

To illustrate a more complex Why-Not question, assume $\mathcal{T} = type(B) \cup type(A)$, where A, B represent the schemas Authors and Books of our running example, respectively. Then, an alternative Why-Not question P over $\mathcal{T}$ is $\mathcal{P} = ((\texttt{B.title:Odyssey}), (true)) \vee ((\texttt{B.author:Sophocles,B.price:}x_1), (x_1 \neq 10 \ \wedge \ x_1 < 20))$.

In the sequel, we will omit the condition (true) for more concise notation, i.e., we may rewrite the c-tuple $(t, (true))$ as t.

Here the reader should remember that, given a query Q whose input schema is $\mathcal{S}_Q$, new attributes (attributes not belonging to the input schema $\mathcal{S}_Q$) have been introduced in the query through join or union specifications. These new attributes are well identified and linked to the input attributes through the renaming ν_Q associated with Q. Answering a Why-Not question requires to track back tuples belonging to the query input instance which is an instance over $\mathcal{S}_Q$. This entails that the c-tuples of the (predicate specifying the) Why-Not question need to be processed to eliminate all new attributes. This translation is done by reversing the query renaming.

Definition 2.7 (Unrenamed predicate w.r.t. a query Q)**.** *Let Q be a query and ν_Q its associated renaming. Let t_c be a c-tuple. Given any $(A_1, A_2, A_{new}) \in \nu_Q$, if $A_{new} \in type(t_c)$, we replace each A_{new} in t_c by A_1, denoted as $\nu_{|1}^{-1}(t_c)$. We proceed analogously for A_2, yielding $\nu_{|2}^{-1}(t_c)$. Then, the mapping $UnR_{(Q,\nu_Q)}$ associates to t_c a predicate defined by:*

1. *if $Q = [R_i]$ (then $\nu_Q = \{\}$) then $UnR_{(Q,\nu_Q)} = t_c$,*
2. *Let Q_1, Q_2 be queries with associated renamings (Q_1, ν_{Q_1}), (Q_2, ν_{Q_2}) respectively.*
   - *if $Q = [Q_1] \bowtie_\nu [Q_2]$, then $UnR_{(Q,\nu_Q)} = UnR_{(Q_1,\nu_{Q_1})}(\nu_{|1}^{-1}(t_c)) \bowtie UnR_{(Q_2,\nu_{Q_2})}(\nu_{|2}^{-1}(t_c))$*
   - *if $Q = [Q_1] \cup_\nu [Q_2]$,, then $UnR_{(Q,\nu_Q)} = UnR_{(Q_1,\nu_{Q_1})}(\nu_{|1}^{-1}(t_c)) \vee UnR_{(Q_2,\nu_{Q_2})}(\nu_{|2}^{-1}(t_c))$*
   - *if $Q = \pi_W[Q_1]$ or $Q = \sigma_C[Q_1]$ (then $\nu_Q = \{\}$) then $UnR_{(Q,\nu_Q)} = UnR_{(Q_1,\nu_{Q_1})}(t_c)$.*

Given $\mathcal{P} = \bigvee_{i=1}^{n} t_c^i$ and the query Q, the urenamed predicate is $\bigvee_{i=1}^{n} UnR_{(Q,\nu_Q)}(t_c^i)$.

Of course, if $\mathcal{P}$ contains only qualified attributes, it is equal to its unrenamed form.

Example 2.3. *Let us modify our sample query Q of Fig. 1, to have an output type $\mathcal{T}_Q$=$\{B.title, A.dob, authorID, publ\}$ (that is, project more attributes at m_Q). The renaming of Q is empty, whereas the renaming associated with its subquery Q_2 (join) is: ν_{Q_2}=$\{(B.aid, A.aid, authorID), (B.pub, A.pub, publ)\}$. Now, let us assume the predicate $\mathcal{P}$=(`B.title:Odyssey, authorID:a1,publ:Harper`). This predicate contains both qualified as well as renamed attributes. More specifically, the attribute `authorID`, can be "unrenamed" to B.aid and to A.aid, which are qualified attributes and cannot be further "unrenamed". In a same way, the attribute `publ` can be "unrenamed" to A.pub and to B.pub. Of course, the qualified attribute `B.title` will remain as is. So, the unrenamed predicate $\mathcal{P}$ is:* (`B.title:Odyssey,B.aid:a1,B.pub:Harper`) $\bowtie$ (`B.title:Odyssey,A.aid:a1,A.pub:Harper`) *which corresponds to the c-tuple:* (`B.title:Odyssey,B.aid:a1,B.pub:Harper, A.aid:a1,A.pub:Harper`).

2.3 Compatibility

Given a Why-Not question in form of a predicate $\mathcal{P}$, we compute a Why-Not answer i.e., a query-based explanation, by tracing source data relevant to the satisfaction of $\mathcal{P}$ through all sub-queries of the query to be analyzed. We identify such relevant data based on the concept of compatibility. We define a tuple to be compatible with either a c-tuple or a predicate.

Definition 2.8 (*c*-tuple compatibility)**.** *Let $\mathcal{I}$ be an instance over a schema $\mathcal{S}$ and t_c be a c-tuple whose set of variables is X and such that $type(t_c) \subseteq \bigcup_{R \in \mathcal{S}} type(R)$*

Then, the tuple t=$(A_1 : v_1, \ldots A_n : v_n) \in \mathcal{I}|_R$, where $R \in \mathcal{S}$ is compatible *with t_c if:*

- *$type(t) \cap type(t_c) \neq \emptyset$ and*
- *$\exists$ a valuation ν : $X \cup adom(t_c) \rightarrow adom(\mathcal{I})$ s.t.*
  - *$\forall A \in type(t_c) \cap type(t) : \nu(t_c.A) = t.A$, with A being an attribute and*
  - *$\nu(t_c) \models t_c.cond$*

Example 2.4. *Given the database schema S and the database instance $\mathcal{I}$ of the running example, and assuming a c-tuple t_c=($(B.title{:}Antigone, B.price{:}v_p), v_p < 50$), the compatible tuple with respect to t_c in $\mathcal{I}$ is: t_1 = (B.bid:b2, B.title:Antigone, B.aid:a2, B.price:49, B.pub:Harper).*

Note that t_1 has two common attributes with t_c and for them, they either share the same constant value (for B.title) or the value of t_1 satisfies the condition of t_c (e.g., a price below 50). As expected, we have no compatible tuples coming from $\mathcal{I}_{Authors}$, as in the c-tuple we have only attributes coming from $\mathcal{I}_{Books}$.

The set of tuples compatible with t_c, called *direct compatible set* with respect to t_c is denoted by Dir^{t_c}. Now, let us consider $\mathcal{S}^{t_c}$ to be the set of relation schemas typing the tuples of Dir^{t_c}. The *indirect compatible set* with respect to t_c, denoted $InDir^{t_c}$, is the restriction of $\mathcal{I}$ on the database schema $\mathcal{S}_Q - \mathcal{S}^{t_c}$. Note that, by definition, $Dir^{t_c} \cap InDir^{t_c} = \emptyset$.

Example 2.5. *Pursuing Example 2.4, Dir^{t_c}=$\{t_1\}$ whereas $InDir^{t_c}$=$\mathcal{I}_{Authors}$.*

2.4 Pickyness

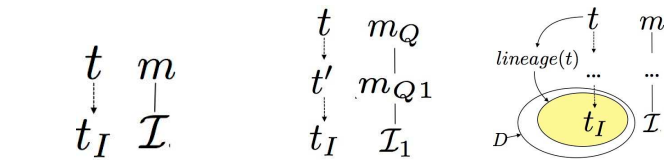
Having introduced the definitions of the Why-Not question and compatibility, this section focuses on the intermediate stage before proceeding to the definition of the Why-Not answers given in Sec. 2.5. Below definitions assume a given query Q, its input schema $\mathcal{S}_Q$ and its input instance $\mathcal{I}_Q$.

Intuitively, given a query Q and a set of compatible tuples (both direct and indirect) in $\mathcal{I}_Q$, our goal is to trace compatible tuples in the data flow of the query tree; that is, identify subqueries of Q that destroy any successor (formally defined below) of these tuples.

To trace compatible tuples through subqueries, we need to process potentially each subquery in Q one after the other. To formalize this procedure, we associate to each subquery Q_i a manipulation m_{Q_i} that serves as a type signature of Q_i. For instance in Fig. 1, subquery Q_1 is associated to m_{Q1} of the form $A \bowtie AB$. The input instance $\mathcal{I}_i$ to a manipulation m_{Qi} includes solely the output of its direct children in the tree (or, in case of leaf nodes, the instance of the corresponding table), e.g. m_{Q_1} and B in Fig. 1 for m_{Q_2}. We denote the output of a manipulation m over its input instance $\mathcal{I}$ as $m(\mathcal{I})$.

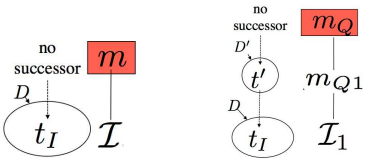
We trace tuples based on *data lineage*, or lineage for short [7], focusing on the lineage of tuples in $m(\mathcal{I})$ w.r.t. m and $\mathcal{I}$. Essentially, lineage describes for any $d \in m(\mathcal{I})$ what is the maximum $D \subseteq \mathcal{I}$ such that: for any $d_\mathcal{I} \in D$, $d \notin m(\mathcal{I} \setminus \{d_\mathcal{I}\})$. Based on the definition of lineage, we say that $t \in m(\mathcal{I})$ *is a successor of* some $t_\mathcal{I} \in \mathcal{I}$, if $t_\mathcal{I}$ is in the lineage of t w.r.t. m. Similarly, we say that $t_\mathcal{I}$ is a *predecessor* of t. Fig. 2(a) illustrates the successor relationship between t and $t_\mathcal{I}$, contained in $m(\mathcal{I})$ and $\mathcal{I}$, respectively.

We now define a tuple successor w.r.t. to a query composed of subqueries, each typed by a manipulation. The definition is illustrated in Fig. 2(b) for the case of unary operators.



(a) w.r.t. a manipulation (b) w.r.t. a query (c) valid successor

Figure 2: Tuple successor t of a tuple $t_{\mathcal{I}}$



(a) Picky manipulation (b) Picky query

Figure 3: Pickyness

Definition 2.9 (tuple successor w.r.t. a query)**.** *Let Q be a query over $\mathcal{S}_Q$ and let $\mathcal{I}$ be an instance over $\mathcal{S}_Q$.*

A tuple $d_Q \in Q(\mathcal{I})$ is a successor of some $d_I \in \mathcal{I}$, with respect to Q if

- *$Q=Op[Q_1]$, where Op is the unary operator σ or π:*
  there exists some $d'_1 \in Q_1(\mathcal{I}_1)$ such that d_Q is a successor of d'_1 with respect to m_Q and either $d'_1=d_I$ or d'_1 is a successor of d_I with respect to Q_1.
- *$Q=[Q_1]Op[Q_2]$, where Op is the binary operator $\bowtie$ or $\cup$, with ν be the renamming associated with Q:*
  there exists some $d' \in Q_1(\mathcal{I}_1) \cup Q_2(\mathcal{I}_2)$ such that d_Q is a successor of d' with respect to m_Q and either $d'=d_I$ or d' is a successor of d_I with respect to Q_1 or with respect to Q_2.

$\mathcal{I}_i$ is the instance over $\mathcal{S}_{Q_i}$ defined by $\mathcal{I}_i=\mathcal{I}\,|_{\mathcal{S}_i}$ for $i=1,2$.

Example 2.6. *Consider the query tree of Fig. 1(c). We reuse the database instance of Fig. 1(b), which includes the tuple d_B=(`b2,Antigone,a2,49,Harper`). The output of Q includes a tuple d_Q=(`Antigone, 400BC`) having lineage(d_Q)={(`b2,Antigone,a2,49,Harper`), (`a2,Sophocles,400BC,Harper`)}. Hence, d_Q is a successor of d_B, justified as follows. When evaluating subquery $Q1$, the tuple d_B survives the selection as its price (equal to 49) is above 20. More specifically, (`b2, Antigone, a2, 49`) $\in Q1(\mathcal{I}_{Books})$. This tuple, denoted d_1, is thus a successor of d_B with respect to $Q1$. The result of $Q2$ in turn includes a tuple d_2=(`b2,Antigone,a2, 49,Harper,Sophocles, 400BC`) that is a successor of d_1 and thus d_B. Similarly, we determine that $d_Q \in Q2(\mathcal{I}_{Q2})$ is a successor of d_B as well.*

Let us now restrict the former notation, to introduce the notion of *valid* successors w.r.t. some set of tuples D. This restriction demands that the lineage of a tuple successor is all in the set D.

Notation 2.1 (Valid successor)**.** *Let Q be a manipulation/query, $\mathcal{I}$ be a well typed input instance for m and $D \subseteq \mathcal{I}$. A tuple $d \in Q(\mathcal{I})$ is a valid successor of some $d_I \in D \subseteq \mathcal{I}$, with respect to Q if d is a successor of d_I w.r.t. Q and lineage(d)$\subseteq D$.*

Fig. 2(c) illustrates the notion of valid successor. From now on, we will generally refer to valid successors when writing successor, unless mentioned otherwise.

The concept of valid successor is used next to define picky manipulations and picky queries, both with respect to a tuple set D and a tuple d s.t. $d \in D$.

When tracing tuples - more specifically, compatible tuples - throughout the query, our goal is to identify which subqueries are responsible for "losing" compatible tuples. These are declared as *picky*, a property at the heart of our definition of Why-Not answers. More specifically, we define picky manipulations and subqueries w.r.t. a tuple set D and a tuple $t_{\mathcal{I}} \in D$. The definitions, given below, are illustrated in Fig. 3.

Definition 2.10 (Picky manipulation)**.** *Let m be a manipulation, $\mathcal{I}$ be a well typed input instance for m and $D \subseteq \mathcal{I}$. Then m is a picky manipulation w.r.t. D and $d_{\mathcal{I}} \in D$, if there is no valid successor d of $d_{\mathcal{I}}$ in $m(\mathcal{I})$.*

Example 2.7. *Consider again the running example, in Fig. 1. Let* $D = \{$(`b3,Aeneid,a3,10,Penguin`), (`a2,Sophocles,400BC,Harper`)$\}$ *and consider the tuple* $d =$ (`a2,Sophocles,400BC,Harper`) $\in D$. *The only successor of* d *w.r.t* m_{Q2} *is*

$$d_{m_{Q2}}=(\texttt{b2,Antigone,a2,49,Harper,Sophocles,400BC})$$

However, the lineage of $d_{m_{Q_2}}$ *also contains the tuple* $d_{\mathcal{I}}=$(`b2,Antigone,a2,49,Harper`) $\notin D$. *Thus,* m *is considered picky w.r.t.* d *and* D.

Definition 2.11 (Picky query)**.** *Let* Q *be a query over* $\mathcal{S}_Q$ *and let* $\mathcal{I}$ *be an instance over* $\mathcal{S}_Q$. *Let also* $D \subseteq \mathcal{I}$ *be a set of tuples. Then* Q *is picky w.r.t.* D *and* $d_I \in D$ *if*

- $Q=Op[Q_1]$, *where* Op *is the unary operator* σ *or* π*:*
  there exists some $d \in D' \subseteq Q_1(\mathcal{I}_1)$ *such that* m_Q *is a picky manipulation w.r.t.* d *and* D', *where* $D'=\{d | d$ *is a valid successor of* d_I w.r.t. Q_1 *and* $D\}$
- $Q=[Q_1]Op[Q_2]$, *where* Op *is the binary operator* $\bowtie_\nu$ *or* $\cup_\nu$*:*
  there exists some $d \in D' \subseteq Q_1(\mathcal{I}_1) \cup Q_2(\mathcal{I}_2)$ *such that* m_Q *is a picky manipulation w.r.t.* d *and* D', *where* $D'=\{d | d$ *is a valid successor of* d_I w.r.t. Q_1 *or* Q_2 *and* $D\}$.

$\mathcal{I}_i$ *is the instance over* $\mathcal{S}_{Q_i}$ *defined by* $\mathcal{I}_i=\mathcal{I}\,|_{\mathcal{S}_i}$ *for* $i=1,2$.

Property 2.1. *Let* Q *be a query over* $\mathcal{S}_Q$ *and let* $\mathcal{I}$ *be an instance over* $\mathcal{S}_Q$. *Let also* $D \subseteq \mathcal{I}$ *be a set of tuples and* $d_{\mathcal{I}} \in D$. *Then, there exists at most one subquery* Q' *of* Q, *s.t.* Q' *is picky w.r.t.* D *and* d_I.

The proof of this property can be done easily, based on the given definitions about a query Q and its input instance $\mathcal{I}_Q$ (Defs 2.2 and 2.3). Based on these, a tuple d may have its origin in one and only one relation instance in the input instance $\mathcal{I}_Q$. Thus, it can follow only one path through its associated subqueries in Q. Let also $D \subseteq \mathcal{I}$ be a set including d. If Q' is picky for d w.r.t. D, then no valid successor can be found after Q'. So, it is certain that there will not exist any valid successor of d in the input of any subquery Q'' of Q , for which Q' is a subquery and consequently no other Q'' can be picky for d.

Example 2.8. *Continuing Example 2.6, consider now* $D=\{d_B, d_A\}$, *where* $d_B =$ (`b2,Antigone,a2,49,Harper`) *(as before) and* $d_A=$(`a3,Virgil,70BC,Penguin`). *We show that the subquery* Q_2 *is picky, both for* d_B *and* d_A *w.r.t.* D.

For d_B, *we stated in the previous example that* d_1 *is its only successor with respect to* $Q1$. *However,* m_{Q2} *is picky for* d_1 *with respect to* $D'=\{d_A, d_1\}$ *as its successor* d_2 *with respect to* m_{Q2} *has not all its lineage in* D'. *Thus,* $Q2$ *is picky for* d_B.

To show that m_{Q2} *is picky for* d_A, *we observe that* $Q2$ *is picky for* d_A, *because there exists no successor of* d_A *with respect to* Q_2.

Picky subqueries in a query Q are considered responsible for pruning certain tuples from the query result.

2.5 The Why-Not Answer

Providing query-based explanations for missing answers is as already stated, the goal of this work. Previously, we defined the Why-Not question to represent the missing aswers identified by the user, w.r.t. some query. In this section, we define the Why-Not answer to represent the explanations returned to the user regarding their Why-Not question.

In order to be accurate and descriptive, but at the same time retain the option for a more general and possibly less confusing (in terms of size) answer, we have worked towards the generation of three kinds of answers; the *detailed*, the *indirect* and the *direct* answers, that form the Why-Not answer. To help the user at understading the definitions of the Why-Not question and the intuition behind them, let us proceed with an example.

Example 2.9. *Assume that in the example of Fig. 1, we add* t_1=(`b0,Odyssey,a1,30,Vintage`) *to Books. Further assuming* $\mathcal{P}$=(`Books.title:Odyssey, Authors.dob:800BC`)*, we find three compatible tuples w.r.t. our predicate* $\mathcal{P}$*, i.e., above tuple* t_1*,* t_2=(`b1,Odyssey,a1,15,Harper`)*, and* t_3=(`a1,Homer,800BC,Penguin`)*. These tuples form the set* D=$\{t_1, t_2, t_3\}$*.*

Using previous definitions, we determine that Q_1 *is picky for* t_2 *while* Q_2 *is picky for* t_1 *and* t_3*, all w.r.t.* D*. Consequently, a possible answer to our Why-Not question defined by* $\mathcal{P}$ *is the set of picky subqueries* $\{Q_1, Q_2\}$*, meaning "*Q_1 *and* Q_2 *are responsible for the exclusion of* (`Books.title:Odyssey, Authors.dob:800BC`) *from the query result". This* direct answer *provides a first hint on where in the query data went missing and is essential for instance if a user has no access to the source instance.*

A more detailed answer *that we can provide using our definitions, specifies the association between a picky subquery and the tuples it has pruned, e.g. "*Q_1 *is picky for* t_2 *while* Q_2 *is picky for* t_1 *and* t_3*". Such information helps during query debugging, as it pinpoints that even though* t_2 *was eliminated at* Q_1*,* t_1 *still exists until* Q_2*, so fixing the query may only require a modification of* Q_2*.*

If now we change Q_1 *to* $\sigma_{B.price>60}$ *and* $\mathcal{P}$ *to* (`A.dob:800BC`)*, we get one compatible tuple, i.e.,* t_3 *and now* $D = \{t_3\}$*. The picky subquery for* t_3 *w.r.t.* D *is again* Q_2*. In this case, however,* $Q_1(\mathcal{I}_{Q_1})$=$\emptyset$*, which makes also* $Q_2(\mathcal{I}_{Q_2})$=$\emptyset$*. Intuitively, even though we know that* Q_2 *is responsible for the exclusion of the desired answer from the result, we suspect that fixing* Q_2 *will not be sufficient.* Q_1 *also has some responsibility for providing an empty join partner, thus could be considered as an* indirect answer*.*

The example above illustrates the different kinds of Why-Not answers we consider. To define them more precisely, in the following we provide the definition for a Why-Not answer.

For the sake of simplicity, we provide below the notion of Why-Not answer for a predicate reduced to one conditional tuple.

Definition 2.12 (Why-Not answer)**.** *Let* Q *be a query and let* t_c *be a c-tuple obtained from unrenaming a Why-Not question over* Q*. The detailed Why-Not answer of* t_c*, denoted* $dW_Q(t_c)$*, is defined by:*

$$\bigcup_{d_\mathcal{I} \in Dir^{t_c}} \{(d_\mathcal{I}, Q') \mid Q' \text{ is a subquery of } Q \text{ and } Q' \text{ is picky w.r.t. } Dir^{t_c} \text{ and } d_I\}$$

The indirect Why-Not answer of t_c*, denoted* $indirW_Q(t_c)$*, is defined by:*

$$\bigcup_{\mathcal{S} \in \mathcal{S}_Q - \mathcal{S}^{t_c^i}} \{Q' \mid Q' \in WQ_\mathcal{S}(t_c) \text{ and } for\ each\ Q'' \in WQ_\mathcal{S}(t_c), Q'' \text{ is a subquery of } Q'\}$$

where

$$WQ_\mathcal{S}(t_c) = \{Q' \mid Q' \text{ subquery of } Q \wedge Q' \text{ picky w.r.t. } InDir^{t_c} \text{ and w.r.t. some } d_I \in InDir^{t_c}_{|\mathcal{S}}\}$$

When analyzing our algorithm, we also consider what we name the *direct* Why-Not answer denoted $dirW_Q(t_c)$ and defined as the union of the detailed answers . More precisely:

$$dirW_Q(t_c)=\{\ Q' \mid (d, Q') \in dW_Q(\mathcal{P})\ \}$$

Recall for the above, that Dir^{t_c} is the direct compatible set w.r.t. t_c and $InDir^{t_c}$ the indirect compatible instance w.r.t. t_c defined over $\mathcal{S}_Q - \mathcal{S}^{t_c}$. Also, note in the $indirW_Q(t_c)$, that Q' is actually the upmost subquery among the $WQ_{\mathcal{S}}(t_c)$.

Defining the Why-Not answer for a general (unrenamed) predicate $\mathcal{P}=\bigvee_{i=1}^{s} t_c^i$ is simply obtained by gathering answers for each t_c^i in one set. For instance, $dirW_Q(\mathcal{P})=\{dirW_Q(t_c^i) \mid i = 1, \ldots, s\}$.[2]

Tab. 1 summarizes the Why-Not answers for the two cases of Example 2.9.

Case	$dirW_Q$	$indirW_Q$	dW_Q
Case1	$\{Q_1, Q_2\}$	$\{\}$	$\{(t_2, Q_1), (t_1, Q_2), (t_3, Q_2)\}$
Case2	$\{Q_2\}$	$\{Q_1\}$	$\{(t_3, Q_2)\}$

Table 1: Direct, indirect and detailed Why-Not answer for Example 2.9

3 The NedExplain Algorithm

So far, we introduced the reader to the problem that we are addressing, i.e., answering why-not questions. We formally defined the notions related to our approach for solving it, setting the foundation for the discussion of our algorithm NedExplain that efficiently answers a Why-Not question.

The general input of our algorithm contains a predicate $\mathcal{P}$ over the output type $type(Q)$ of a query Q over a database schema $\mathcal{S}_Q$, represented as a query tree T and executed on a query input database instance $\mathcal{I}_Q$. We describe the algorithm in two sections. First, we describe necessary preprocessing before we discuss the internals of the algorithm.

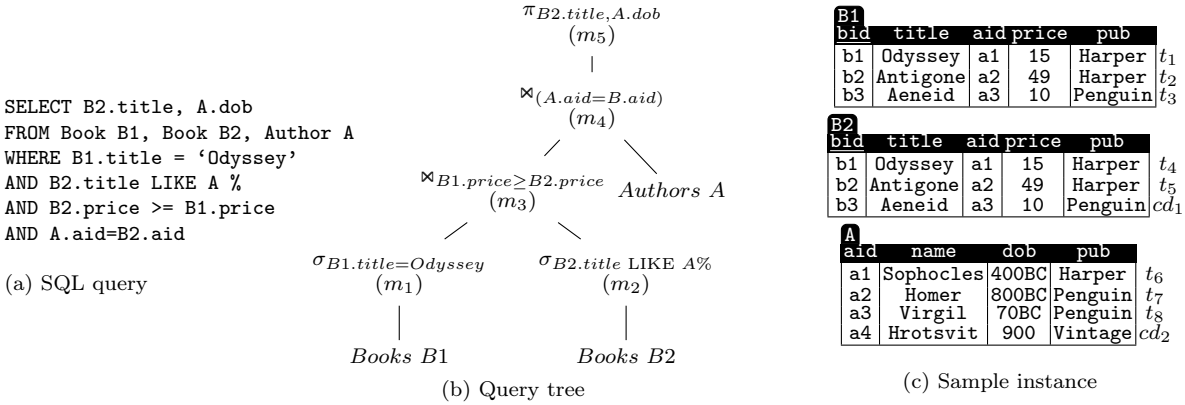
3.1 Preprocessing

Preprocessing includes the unrenaming of the predicate forming the Why-Not question, which is performed once. Then, for each c-tuple t_c in $\mathcal{P}$ the preprocessing regards the computation of the compatible set of tuples w.r.t. t_c and the initialization of a primary global structure, named Tab_Q as well as some secondary global structures. We discuss each preprocessing individually below.

Unrenaming. First step before proceeding to the algorithm, is to unrename $\mathcal{P}$, as defined in Def. 2.7. By this process, we obtain $unR(\mathcal{P})=\bigvee_{i=1}^{n} t_c^i$, where every t_c^i contains only qualified attributes from the output query type and the unrenaming procedure.

We continue by repeating the following procedures regarding one $t_c \in unR(\mathcal{P})$ at a time. This implies also executing the whole procedure described in Section 3.2 for each c-tuple. Thus, in the following, we generally refer to a c-tuple t_c. The set of the results from the execution for all c-tuples provides the overall answer w.r.t. $\mathcal{P}$.

[2]Notice that $dirW_Q(\mathcal{P})$ is a set of sets.



(a) SQL query

(b) Query tree

(c) Sample instance

Figure 4: SQL query (a), query tree (b), and instance (c) of example of Section 3

CompatibleFinder. From t_c, we can easily compute the compatible set of tuples $C \subseteq \mathcal{I}_Q$ w.r.t. t_c, by performing appropriate SELECT statements that retrieve ids[3] of the relations referenced by the qualified attributes of t_c (as illustrated in Example 3.1). Note that, we demand that all (attribute:value) pairs in t_c that reference the same relation must co-occur in the same tuple, also illustrated below.

Primary global structure $\mathbf{Tab_Q}$. The input of the algorithm is organized by means of a main global structure, denoted as Tab_Q, accessible from any part of the algorithm. Tab_Q is used to store intermediate computations as well, as discussed later. More specifically, Tab_Q contains the following labeled entries for each subquery m of Q.

- *Input*: the input tuple set of the subquery m
- *Output*: the output tuple set of the subquery m
- *Compatibles*: the set of tuples
  $\{d_i | d_i \in m.Input \wedge (d_i \in C \vee d_i \text{ is a successor of some } d \in C \text{ w.r.t. } m)\}$
- *Level*: the depth of m in T
- *Parent*: the parent node (subquery) of m in T
- *Op*: the root operation of m

To denote an entry e of a subquery m, we write $m.label(e)$, where $label(e)$ is the label of e.

Initialization is trivial for $m.Op$, $m.Parent$ and $m.Level$ based on T. For $m.Input$, initialization is possible for any m that is a base relation, based on $m.Input=\{\mathcal{I}_Q|_{R_i}\}$ *where* $m=[R_i], R_i \in \mathcal{S}_Q$. Then, we initialize $m.Compatibles$ for any m that is a base relation by $m.Compatibles=\{C^{t_c}|_{R_i}\}$ *where* $m=[R_i], R_i \in \mathcal{S}_Q$. The rest of the entries get updated during the execution of the algorithm. In order to efficiently access the information in Tab_Q that is necessary during processing, subqueries are stored in order of decreasing depth ($m.Level$) in the query tree. We access subquery m at position i using the notation $m = Tab_Q[i]$.

Example 3.1. *Let us demonstrate how Tab_Q is initialized based on the query Q provided in Fig. 4(a)[4]). Books and Authors retain the same schema as previously and the database instance I_Q is shown in Fig. 4(c).*

[3]These queries assume that each table has a key attribute to uniquely identify a tuple. If no such key exists, the queries can be modified to retrieve the complete tuples, which then need to be stored in main memory for further processing.

[4]Based on our renaming model, the θ-join is actually a join with empty renaming followed by a selection. We however use this condensed representation to simplify the discussion.

The answer to the above query includes the two tuples {(`B2.title:Antigone,A.dob:800BC`) *and* (`B2.title:Aeneid,A.dob:70BC`)}.
Let us now assume a predicate $\mathcal{P}$ *that includes one c-tuple* t_c=((`B2.title:Aeneid, A.dob:900`), *true*). *Note that it is already in its unrenamed form. The compatible tuple set of* t_c *is computed by executing the SQL given in Fig. 5. We thus obtain the set of compatible tuples* C=$\{cd_1, cd_2\}$ *where* cd_1=`B2(b3,Aeneid,a3,$10,Penguin)` *and* cd_2=`A(a4,Hrotsvit,900,Vintage)` *(also see tuple labels in Fig. 4(c)). This results in* $C\,|_A$=$\{cd_2\}$ *and* $C\,|_{B2}$= $\{cd_1\}$. *Obviously,* $C|_{B1}$=$\emptyset$.

Using the information gathered so far, we initialize Tab_Q *as illustrated in Tab. 2.*

Secondary global structures. Apart from Tab_Q, we make use of some other global structures, which are:

- $EmptyOutputMan$: the set of subqueries producing the empty set
- $Non\text{-}PickyMan$: the set of non-picky subqueries
- $PickyMan$: the set of pairs (m, blocked), where m is a Picky subquery and blocked=$\{d | d \in m.Input \wedge$ m is picky for $d\}$

Note that the set of subqueries $EmptyOutputMan$ relate to the indirect Why-Not answer. Likewise, the $PickyMan$ set will be used to form the direct Why-Not answer and the detailed Why-Not answer.

3.2 Computing Why-Not Answers

Having defined the prerequisites for the computation of the Why-Not answers, we now proceed with the algorithmic solution. The main idea is to visit the subqueries of Q in the order given by Tab_Q, which represents the descending depth in the query tree. At each step, we identify the successors of the compatible tuples and keep track of the *Non-Picky* and *Picky* subqueries along the way. In the end, we provide all kinds of Why-Not answers (detailed, direct, indirect) in the common structure named $ANSWERS$.

Alg. 1 is the main algorithm of NedExplain. It takes as input one c-tuple t_c, so it will be executed once per each t_c in $UnR(\mathcal{P})$. The compatible tuple set C as well as the global structures are initialized w.r.t. t_c in the start of each execution. The overall answer w.r.t. $\mathcal{P}$, will then be the set of the generated $Answers$.

After the necessary preprocessing, the algorithm potentially iterates through all the subqueries stored in Tab_Q. Nevertheless, we expect that it terminates before reaching the end of Tab_Q. This early termination is verified by $checkEarlyTermination$ (Alg. 3) called at line 5. If $checkEarlyTermination$ returns true, we compute the detailed, the direct and the indirect Why-Not answers (Algs. 4 and 5, respectively), which are both returned in the structure $ANSWER$. If $checkEarlyTermination$ returns false, we continue with the evaluation of the

```
SELECT B2.id
FROM Book2 B2
WHERE B2.title = 'Aeneid'


SELECT A.id
FROM Author A
WHERE A.dob = '900'
```

Figure 5: Executed SQL statements

Entry label	**B1**	**B2**	$\mathbf{m_1}$	$\mathbf{m_2}$	$\mathbf{m_3}$	**A**	$\mathbf{m_4}$	$\mathbf{m_5}$
Input	$\mathcal{I}_{\|B1}$	$\mathcal{I}_{\|B2}$				$\mathcal{I}_{\|A}$		
Compatibles	$\emptyset$	$C_{\|B2}$				$C_{\|A}$		
Output								
Level	4	4	3	3	2	2	1	0
Parent	m_1	m_2	m_3	m_3	m_4	m_4	m_5	
Op	relation schema	relation schema	σ	σ	$\bowtie$	relation schema	$\bowtie$	π

Table 2: Primary global structure Tab_Q

Algorithm 1: NedExplain

```
   Input: S_Q, Q, I_Q, t_c over type(Q)
   Output: ANSWER
1  CompatibleFinder;
2  Initialiazation of global structures:
   Tab_Q, Non − PickyMan, EmptyOutputMan, PickyMan,
   StacksList;
3  for (int i=0,..., Tab_Q.size()-1) do
4      m=Tab_Q[i];
5      if (checkEarlyTermination(m)) then
6          ANSWER.Detailed←DetailedAnswer().Detailed;
7          ANSWER.Direct←GeneralAnswer().Direct;
8          ANSWER.Indirect←GeneralAnswer().Indirect;
9          return ANSWER;
10     m.Ouput← m(m.Input);
11     if (m.Ouput=∅) then
12         EmptyOutputMan← EmptyOutputMan ∪{m};
13     p←m.Parent;
14     p.Input←p.Input ∪ m.Output ;
15     if (m.Op='relation schema') then
16         if (m.Compatibles≠ ∅) then
17             p.Compatibles ← p.Compatibles
               ∪m.Compatibles;
18             Non-PickyMan←Non-PickyMan∪{m};
19     else
20         p.Compatibles ← p.Compatibles ∪
           FindSuccessors (m);
21 return null;
```

Algorithm 2: *FindSuccessors*

```
   Input: m, a subquery
   Output: the successors of tuples in m.Compatibles, in
           the output of m
1  compOrigins← {m'|(m'.Output ∩ m.Compatibles ≠
   ∅) ∧ (m'.parent = m)};
2  successors← ∅;
3  foreach o ∈ m.Output do
4      if lineage(o) ⊆ Dir^{t_c} ∪ InDir^{t_c} then
5          successors ← successors ∪(lineage(o) ∩ Dir^{t_c});
6  Blocked ← m.Compatibles \ successors;
7  if (successors ≠ ∅) then
8      Non-PickyMan ← Non-PickyMan ∪{m};
9  if (Blocked ≠ ∅ ∧ v) then
10     PickyMan ←PickyMan ∪ {(m,Blocked)};
11 return successors;
```

Algorithm 3: checkEarlyTermination

```
   Input: m, a subquery
   Output: earlyTermination, TRUE indicates the early
           termination of the procedure
1  earlyTermination← TRUE;
2  i←poisition of m in Tab_Q;
3  if (i≠0 ∧ m.Level≠Tab_Q[i-1].Level) then
4      int j=i-1;
5      while
       (earlyTermination∧j≥0∧Tab_Q[j].Level=Tab_Q[i-
       1].Level))
       do
6          if (Tab_Q[j] ∈ Non-PickyMan) then
7              earlyTermination=FALSE;
8          j←j-1;
9      if (earlyTermination) then
10         while (earlyTermination∧i < Tab_Q.size()) do
11             if (Tab_Q[i].Op='relation schema') then
12                 earlyTermination←FALSE;
13             i ← i + 1;
14 else
15     earlyTermination←FALSE;
16 return earlyTermination;
```

Algorithm 4: General Answer

```
   Output: Direct, Indirect
1  Indirect ← ∅ ;
2  Direct ← ⋃_{0≤i≤|PickyMan|} {m_i|(m_i, Blocked_i) ∈
   PickyMan};
3  forall the (m subqueries ∈ Direct) do
4      if (m ∈ EmptyOutputMan ∧ m.Children ∩
       EmptyOutputMan ≠ ∅) then
5          Indirect ← Indirect ∪ (m.Children ∩
           EmptyOutputMan);
6  return Direct, Indirect;
```

Algorithm 5: Detailed Answer

```
   Output: Detailed, a set of pairs of the form (picky
           subquery,compatible tuple)
1  Detailed ← ∅;
2  PM ← ⋃_{0≤i≤|PickyMan|} {m_i|(m_i, Blocked_i) ∈ PickyMan};
3  forall the (m subqueries ∈ PM) do
4      forall the
       (d ∈ Blocked, where(m, Blocked) ∈ PickyMan) do
5          Detailed ← Detailed ∪ {(m, d)};
6  return Detailed;
```

subquery on its input (line 10) and maintain the entries of the subquery's parent p in Tab_Q. We also maintain the secondary global structures $EmpytOutputMan$ and $Non\text{-}PickyMan$.

For all subqueries except for those that correspond to relation schemas (as identified by $m.Op$), *FindSuccessors* (Alg. 2) is called in order to find possible successors of compatible tuples in the output of the current subquery. Simultaneously, we maintain all the secondary global structures and update the entries of the parent subquery p with the generated results. Note that the procedure in *FindSuccessors* is trivial for base subqueries (relation schemas).

Let us now discuss the different sub-algorithms called by Alg. 1 in more detail.

checkEarlyTermination (Alg. 3). To decide whether or not we have all information in hand to compute our Why-Not answer, *checkEarlyTermination* evaluates the position of the node

representing the input subquery m in the query tree relative to the position of compatible tuples and their successors at other subqueries. More specifically, if m is the leftmost node at some level in the query tree T, we perform two checks: 1) we check if in the former level we have any $NonPicky$ subqueries and if not, 2) we also check if among the rest of the subqueries there exists one with type 'relation schema'. We stop the procedure if there are no more compatible tuples to trace from m on.

Finding and managing successors (Alg. 2). First, recall that when we are talking about successors, we imply valid successors, unless differently stated. Alg. 2 checks for every tuple in the output of m, if it has its lineage all in the set $Dir^{t_c} \cup InDir^{t_c}$. In this case, o is a (valid) successor of the compatible tuples that are both in the lineage of o and in the compatible tuples set in the input of m. If we find no successors of some $d \in m.Compatibles$, we mark d as blocked and keep the association with the subquery in the $PickyMan$ structure. This means that at this point, we have found one member of the Why-Not detailed answer. If we have found successors for at least one d in the compatible input, we mark the subquery as $NonPicky$. The set $\bigcup_{j=1}^{l} S_j$, (where l is the size of the compatible tuple set in the input of the subquery) is returned to Alg. 1 to be added in the parent subquery's compatible input.

Computing the Why-Not answer (Algs. 4 and 5). Alg. 4 computes the direct and indirect Why-Not answer s. The direct one consists of all the subqueries that were marked as $Picky$, while the indirect contains those subqueries that produced the empty set as result, and were possible culprits for the blocking of a successor at some subquery. The Detailed Answer, computed in Alg. 5 consists of all the (*compatible tuple*, *picky subquery*) pairs computed by the structure $PickyMan$.

We have decided in our alogorithm, to return the set of Why-Not answers, as defined in the former section. However, further post-processing steps can be added to this point, depending on the application domain. For example, one could furtherly order the answers w.r.t. assigned weights, that depend for example on the number of *picked* tuples at the picky subqueries, or the position of the subquery on the query tree (e.g. *frontier* subquery), or even exclude the Detailed Answer when it provides no further information.

Sample run of Alg. 1. To aid the reader follow the algorithm, we now return to the running example of this section and continue with the computation of the Why-Not answers by the given algorithm and regarding also the initialized global structures introduced previously. Tab. 3 summarizes the generated results during this procedure and is actually an abstraction of Tab_Q. During the execution of Alg. 1, a new row is created in this table for each new loop, thus for each next subquery in Tab_Q, until the algorithm exits with an "early termination". For a clarification on the generated results, consider the following indicative cases:

- row 2(m=$Book\_b$): This row concerns the Book_b subquery, which has its m.Input and m.Compatibles initialized by the given information. Alg. 3 does not signal an early termination, since m is not the first subquery that has m.Level=1, so the execution of Alg. 1 continues. Trivially, here m.Output=m.Input. The parent subquery is m_2; so, m_2.Input and m_2.Compatibles get initialized with m.Output and m.Compatibles, respectively. Moreover, since no compatible tuples were blocked, m is classified as $Non-Picky$.
- row 7(m=m_4): Alg. 1 filled the previous rows of the table in previous iterations. For the current row, m.Input is filled in with the values from the outputs of Author and m_3, whereas m.Compatibles is filled with cd_2 from Author and the successor of cd_1 from m_3 (denoted $cd_1 \bowtie t_1$). Alg. 3 does not signal an early termination, since both Author and m_3 (former level subqueries) are non-picky subqueries. So, Alg. 1 continues with the evaluation of m

m	m.Input	m.Output	m.Compatibles	m.Blocked
Book_a	$\mathcal{I}_{\|Book\_a}$	$\mathrm{I}_{\|Book\_a}$	$\emptyset$	$\emptyset$
Book_b	$\mathrm{I}_{\|Book\_b}$	$\mathrm{I}_{\|Book\_b}$	cd_1	$\emptyset$
m_1	$\mathrm{I}_{\|Book\_a}$	t_3	$\emptyset$	$\emptyset$
m_2	$\mathrm{I}_{\|Book\_b}$	t_6,cd_1	cd_1	$\emptyset$
m_3	t_3, t_6, cd_1	$t_3 \bowtie t_6, t_3 \bowtie cd_1$	cd_1	$\emptyset$
Author	$\mathrm{I}_{\|Author}$	$\mathrm{I}_{\|Author}$	cd_2	$\emptyset$
m_4	$t_3 \bowtie t_6, t_3 \bowtie cd_1, \mathrm{I}_{\|Author}$	$t_3 \bowtie t_6 \bowtie t_8, t_3 \bowtie cd_1 \bowtie t_{10}$	$cd_2, t_3 \bowtie cd_1$	$cd_2, t_3 \bowtie cd_1$
m_5	$t_3 \bowtie t_6 \bowtie t_8, t_3 \bowtie cd_1 \bowtie t_{10}$		$\emptyset$	

Table 3: Example Results

on m.Input and fills the entries m.Output, as well as the parent's m_5.Input accordingly. Continuing with the call to Alg. 2, we conclude that m is a picky subquery and that it has blocked all the tuples in m.Compatibles (m.Blocked=m.Compatibles), which means that no successors have survived this subquery. Note here, that even though there exists one tuple in m_4.Output ($t_1 \bowtie cd_1 \bowtie t_8$) that contains the compatible tuple cd_1 in its lineage, it is not a valid successor of cd_1 since it is not also a successor of cd_2. So, m_5.Compatibles=$\emptyset$. Fig. 6 shows the state of secondary global structures at this point.

- row 8(m=m_5): In this row, m.Input and m.Compatibles got their values from the previous step. The call to Alg. 3 marks the early termination of the algorithm; m_5 is the first subquery having m.Level=0 and m_4, which is the only subquery in the previous level, is a picky subquery. Moreover, there are no upper subqueries that could contain some compatible tuples. So, Alg. 1 terminates by calling Algs. 4 and 5, each of which provide the respective part of the Why-not Answer: Answer.detailed= $\{(cd_2, m_4),(cd_1, m_4)\}$ and Answer.direct=$\{m_4\}$. Note here, that m_4 is a picky subquery also for cd_1 because it is picky for its successor ($t_1 \bowtie cd_1$).

The Nedexplain algorithm, is designed to produce correct answers w.r.t. the definitions provided in Sec. 2. Also, in the scope of the given query tree representation, this set of answers is complete. In the future, we are aiming at making our algorithm independent from the query tree representation, i.e., provide the complete set of Why-Not answers which is indifferent to the query plan.

4 Experiments

In the former sections we have introduced NedExplain, a new approach for calculating *Why-Not answers*. As reviewed in Section 5, we have identified a set of drawbacks of the state-of-the-art approach Why-Not [3] to calculate *Why-not answers* based on the query. We now compare NedExplain and Why-Not to each other, demonstrating that NedExplain is able to produce comparable or better results both in terms of efficiency and answer quality.

Non-PickyMan
A
m_3
m_2
$B2$

EmptyOuputMan

PickyMan	
subquery	*Blocked tuples*
m_4	$cd_2, t_1 \bowtie cd_1$

Figure 6: Global structures after m_4

4.1 Setup and Use Cases

We have implemented *NedExplain* and *Why-Not* (actually, its bottom-up version as it most resembles the approach of NedExplain and is the more practical approach) using Java, based on source code the authors of Why-Not kindly provided to us.The implementation of Why-Not makes use of the lineage tracing capabilities of the Trio system (http://infolab.stanford.edu/trio/), which we also used in our implementation. The experiments have been performed on an Oracle Virtual Machine running Windows 7 and using the 2GB out of the 4GB of main memory of a Mac Book Air with 1.8 GHz Intel Core i5, running MAC OS X 10.8.3. We used PostgreSQL 9.2 as database.

Our datasets originate from three databases named *crime*, *imdb*, and *gov*. The *crime* database corresponds to the sample crime database of Trio and was previously used to evaluate Why-Not. The data describes crimes and involved persons (suspects and witnesses). The *imdb* database is built on real-world movie data extracted from IMDB (http://www.imdb.com) and MovieLens (http://www.movielens.org). Finally, the *gov* database contains information about US congressmen and financial activities (collected at http://bioguide.congress.gov, http://usaspending.gov, and http://earmarks.omb.gov). The size of the relations in the databases ranges from 89 to 9341 records, with *crime* being the smallest and *gov* the largest database. For abbreviation, in the following discussion each relation instance is referred to by its initials, for example M refers to the Movies instance and L to the Locations instance. Moreover, when multiple instances of some relation are needed, we distinguish them by numbers, e.g., *M1* and *M2*.

For each database, we have created a series of use cases (see Tab. 4 (b)). Each use case consists of a query further defined in Tab. 4(a) and a Why-Not question in form of a predicate $\mathcal{P}$.

Table 4 displays a summary of the use cases. The prefix of each use case indicates the database against which the query is executed. The *Predicate* column represents the Why-Not question w.r.t the query identified in the *Query* column. As stated in the definitions in Section 2.2, the predicate contains (disjunctions of) $(attribute, value)$ pairs where *attribute* belongs to the type of the query result. In all use cases, we assume the condition always being *true*. For example, the predicate (`W.Name:Susan, C2.Type:kidnapping`) in *Crime7* represents literally the question: *"Why is there not a result tuple with W.Name=Susan and C2.Type=Kidnapping?"*. In some cases, we consider more than one predicate for a given query, which allows us to pinpoint the differences between the two algorithms.

The queries have been designed to include simple (Q4,Q6) and more complicated (Q1, Q3, Q5, Q7) queries, queries containing self-joins (Q3, Q4) and queries having empty intermediate results (Q2).

Based on the use cases above, we now evaluate NedExpain and Why-Not, both in terms of answer quality and efficiency.

4.2 Answer Quality

When running all our use-cases using both Why-Not and NedExplain, we obtain the why-not answers summarized in Table 5. For NedExplain, we distinguish among the detailed, the direct and the indirect Why-Not answer.

At first sight, the answers provided by Why-Not are simpler and clearer; they generally consist of a small number of subqueries. On the other hand, NedExplain provides an answer more complex in structure, but also richer in terms of information provided. Notice that the direct answer resembles the answer returned by Why-Not, thus providing an "easy-consumable"

Query	Expression
Q1	$\pi_{P.name,C.type}(C \bowtie_{sector} W \bowtie_{witnessName} S \bowtie_{hair,clothes} P)$
Q2	$\pi_{P.name,C.type}((\sigma_{C.sector>99}(C)) \bowtie_{sector1} W \bowtie_{witnessName} (S) \bowtie_{hair,clothes} P)$
Q3	$\pi_{W.name,C2.type}(W \bowtie_{sector2} C2 \bowtie_{sector1} \sigma_{C.type=Aiding}(C))$
Q4	$\pi_{P2.name}(\sigma_{P1.name \neq P2.name}(P2 \bowtie_{hair} (\sigma_{P1.name<B}(P1))))$
Q5	$\pi_{name,L.locationid}(L \bowtie_{movieId} ((\sigma_{M.year>2009}(M)) \bowtie_{name} (\sigma_{R.rating \geq 8}(R))))$
Q6	$\pi_{Co.firstname,Co.lastname}(\sigma_{AA.party=Republican}(AA)) \bowtie_{id} (\sigma_{Co.Byear>1970}(Co)))$
Q7	$\pi_{SPO.sponsorId,SPO.sponsorln,E.camount}(E \bowtie_{earmarkId} (\sigma_{ES.substage=Senate\ Committee}(ES) \bowtie_{id} (\sigma_{SPO.party=Republican}(SPO))))$

Use case	Query	Predicate
Crime1	Q1	(P.Name:Hank,C.Type:Car theft)
Crime2	Q1	(P.Name:Roger,C.Type:Car theft)
Crime3	Q2	(P.Name:Roger,C.Type:Car theft)
Crime4	Q2	(P.Name:Hank,C.Type:Car theft)
Crime5	Q2	(P.Name:Hank)
Crime6	Q3	(C2.Type:kidnapping)
Crime7	Q3	(W.Name:Susan,C2.Type:kidnapping)
Crime8	Q4	(P2.Name:Audrey)
Imdb1	Q5	(name:Avatar)
Imdb2	Q5	(name:Christmas Story,L.locationId:USANew York)
Gov1	Q6	(Co.firstname:Christopher)
Gov2	Q6	(Co.firstname:Christopher,Co.lastname:MURPHY)
Gov3	Q6	(Co.firstname:Christopher,Co.lastname:GIBSON)
Gov4	Q7	(sponsorId:467)
Gov5	Q7	((SPO.sponsorln:Lugar,E.camount:x),x=1000)

Table 4: Queries (a),and Use cases (b)

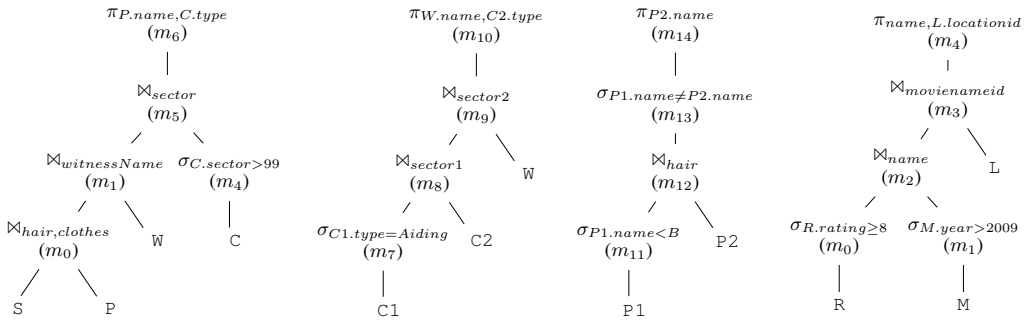
		NedExplain Answers		
Use case	Why-Not	Detailed	Direct	Indirect
Crime1		$(P.Id{:}2, m_2), (C.Id{:}2, m_2)$	m_2	
Crime2	m_0	$(P.Id{:}604, m_0), (C.Id{:}2, m_2)$	m_0, m_2	
Crime3	m_0,m_4	$(P.Id{:}604, m_0), (C.Id{:}2, m_4)$	m_4, m_0	
Crime4	m_4	$(P.Id{:}2, m_5), (C.Id{:}2, m_4)$	m_4, m_5	
Crime5	m_4	$(P.Id{:}2, m_5)$	m_5	m_4
Crime6	m_7	$(C2.Id{:}396, m_8), (C2.Id{:}85, m_8), \ldots, (C2.Id{:}112, m_8),$	m_8	
Crime7	m_7	$(C2.Id{:}396, m_8), (C2.Id{:}85, m_8), \ldots, (C2.Id{:}112, m_8), (W.Id{:}2, m_9)$	m_8, m_9	
Crime8		$(P2.Id{:}51, m_{12})$	m_{12}	
Imdb1	m_1	$(R.Id{:}124, m_2), (M.Id{:}18, m_1)$	m_1,m_2	
Imdb2		$(L.Id{:}1, m_3), (M.Id{:}4, m_3), (R.Id{:}245, m_3)$	m_3	
Gov1	m_2	$(Co.Id{:}569, m_0), (Co.Id{:}1495, m_0), (Co.Id{:}1072, m_2), (Co.Id{:}772, m_0)$	m_0, m_2	
Gov2	m_1	$(Co.Id{:}1072, m_2)$	m_2	
Gov3	m_0	$(Co.Id{:}569, m_0)$	m_0	
Gov4	m_4	$(SPO.Id{:}9, m_4), (ES.Id{:}80, m_8), (ES.Id{:}78, m_8), (ES.Id{:}79, m_8)$	m_4, m_8	
Gov5	m_6	$(E.Id{:}15, m_6), (E.Id{:}324, m_6), \ldots, (E.Id{:}533, m_6), (SPO.Id{:}199, m_6)$	m_6	

Table 5: Why-Not and NedExplain answers, per use case

answer as well. Still they are not the same, as we will describe later in this section. The following discussion highlights the relevancy of each Why-Not answer NedExplain returns with respect to each other as well as with respect to results the Why-Not algorithm returns.

Detailed vs. direct and indirect Why-Not answers. Consider the *Crime6* use case and refer also to the associated query tree ($Q3$) in Fig. 7(b). The direct answer (but not the Why-Not algorithm answer, as described later) indicates that m_8 is a picky subquery. Next, consider the Detailed answer; it consists of 11 pairs of the form $(cd_i, m_8), i = 1, \ldots, 11$. Even though here we are not precise about which the compatible tuples cd_i are, the answer is evident; m_8 is the responsible subquery. Despite the size of the detailed answer, we do not gain any additional useful information compared to the direct one. Thus in this case, the direct answer is equally informative and at the same time less overwhelming.

However, this is not always the case. More specifically, the simplicity of the direct answer may hide from the user more specific, but essential information, for example in cases where the answer is not a single picky subquery like in *Crime7*. Here, the direct answer (again not the same as in Why-Not as explained later) identifies two picky subqueries, i.e., m_8 and m_9 (refer to $Q3$ in Fig. 7(b)). From the detailed answer, we moreover obtain the knowledge that there were eleven tuples (originating from Co) for which m_8 is picky, but also one tuple (originating from W) for which m_9 is picky. This information can be useful, as it indicates that no valid successors of compatible crime tuples reached m_9 to join with valid witness tuples. So, the existence of a



(a) Query $Q2$ (Crime3, Crime4, Crime5) (b) Query $Q3$ (Crime6, Crime7) (c) Query $Q4$ (Crime8) (d) Query $Q5$ (Imdb1, Imdb2)

Figure 7: Query trees for queries $Q2$, $Q3$, $Q4$, $Q5$

more detailed answer can be of major help towards the understanding the "background" of the result.

NedExplain vs. Why-Not. Our first comparison between NedExplain and Why-Not focuses on use case $Crime5$, with the associated query $Q2$ (its query tree is given in Fig. 7(a)) having an intermediate empty result on m_4 ($\sigma_{sector>99}(C)$). The Why-Not algorithm identifies m_4 as a picky subquery, which is correct in the sense of responsibility for the missing result tuple. It cannot yet be considered as picky in the strict form, for not blocking directly any compatible tuple. In this use case, NedExplain provides a more complete and descriptive answer. It identifies m_5 as a picky subquery, and in addition it includes m_4 in the indirect answer. Knowing that m_4 produced an empty set right before the join in m_5 can possibly be another reason for m_5 being identified as picky.

Let us now focus on the cases where the answers of Why-Not and NedExplain differ that we already mentioned previously, i.e., $Crime6$ and $Crime7$. Both use cases relate to $Q3$, which contains a self join on relation Crime. The Why-Not algorithm falsely identifies m_7 ($\sigma_{C1.type=Aiding}(C1)$) as a picky subquery, because it locates the compatible tuples (unpicked data items in Why-Not algorithm) in both $C1$ and $C2$. So, as a result the compatible tuples from $C1$ with $type$:$Kidnapping$ are naturally picked at m_7. This problem is solved by our algorithm, by introducing the notion of qualified attributes. In this way, we locate the compatible tuples only in the correct instance of the relation Crime, $C2$, according to the type of the output of the query $Q3$.

Another problem having its origin in the identification of compatible tuples can be spotted at use case $Crime8$. Even though it is based on a very simple query (refer to $Q4$ in Fig. 7(c)), the Why-Not algorithm finds no answers at all. $Q4$ searches for persons that have the same hair as persons whose names start with a letter smaller than B (while not being the same person). The compatible source tuples are both identified in P1 and P2. The one coming from P2 does not find any join partners in m_{12} from P1 as the only candidate ones have names starting with C or D, namely Davemonet, Chiardola, and Debye. So m_{12} is picky for the compatible tuple coming from P2. The one coming from P1 survives the selection of m_{11} and joins with the three persons with equal hair color coming from P2, namely Davemonet, Chiardola, and Debye. Hence, m_{12} is not picky for three successors of the compatible tuple originating from P1, and it is easy to verify that the same is true for the remaining subqueries to be processed. Hence, Why-Not believes that Audrey is actually not missing from the result.

NedExplain on the other hand will correctly locate the compatible tuple only in P2. As mentioned previously, all candidate join partners coming from P1 for Audrey (that comes from P2), will be discarder in m_{11} making m_{12} a picky subquery for the associated compatible tuple

($P2.Id$:51).

Finally, we review use case $Imdb2$ where the associated predicate $\mathcal{P}$ is not in its unrenamed form. This means that it contains attributes that are not in $\mathcal{S}_Q$, but instead were introduced through renamings associated to the subqueries. This use case is based on query $Q5$ (see Fig. 7(d)), with the following renaming associated to its subquery m_2 (join): ν= ($R.moviename, L.moviename, name$). The considered predicate $\mathcal{P}$=(`name:Christmas Story,Lor.locationId:USANew York`) refers to the new attribute *name* associated with the subquery $m_0 \bowtie m_1$ in $Q5$. Thus, before running NedExplain, we transform $\mathcal{P}$ to its unrenamed form: (`R.name: Christmas Story, Lor.locationId: USANew York`) $\bowtie$ (`L.name: Christmas Story, Lor.locationId: USANew York`) = (`R.name: Christmas Story, L.name: Christmas Story, Lor.locationId: USANew York`). As we said, for the compatible tuples we calculate *valid successors* to trace in the query tree. This method leads us to the identification of m_3 as a picky subquery, i.e. the subquery after which we find no more valid successors. Why-Not on the contrary relies on tracing successors (not necessarily valid) of the compatible tuples, which in this case still can be found in the result set. So, no picky subqueries are identified and no answers are returned, even though none of the successors of the compatible tuples corresponds to the tuple of interest.

Overall, in this preliminary study on answer quality, we observe that NedExplain produces a correct answer for all use cases, as opposed to Why-Not that showcases no, imprecise, or incomplete results in some cases. Furthermore, the different types of answers NedExplain returns convey more information than answers returned by the Why-Not algorithm, potentially improving a developer's analysis and debugging experience. We will further analyze result quality in a more thorough evaluation in the future.

4.3 Runtime Evaluation

Next, we study the runtime behavior of computing Why-Not answers when using NedExplain or Why-Not.

Fig. 8 displays, for each use case, the time (in ms) each algorithm needs to produce its Why-Not answers. Generally, we observe that NedExplain is faster compared to Why-Not. One reason is that the implementation of the Why-Not algorithm requires the usage of Trio for lineage calculation. Apparently, the dependence of Why-Not on Trio makes the algorithm slower, especially when many trio tables are referenced as in $Crime1$ and $Crime2$. NedExplain traces the compatible tuples by issuing queries directly to the underlying Postgres database based on their unique identifiers in order to find their successors, which speeds up the process.

In several cases (e.g, $Gov4$ and $Gov5$) the gain of performance by this implementation makes up for the additional time NedExplain spends to check the condition that a successor is actually a valid successor, still yielding a comparable performance to Why-Not. Indeed, while in the Why-Not algorithm, it is sufficient to have a successor of a compatible tuple in order to continue tracing it, in NedExplain we further impose the restriction for the successor to have its lineage in the respective compatible tuple set. As explained before, this distinction accounts for the more accurate Why-Not answers NedExplain produces, so it is time well spent.

In the future, we plan to more extensively study the impact of various parameters on runtime, including the size of the database, the size of the query, the size and distribution of compatible tuples. We will also measure the effect of using materialized or non-materialized views for the initial query, on the algorithm's efficiency. However, this first set of experiments on runtime indicates that NedExplain is able to provide high-quality Why-Not answers in reasonable time.

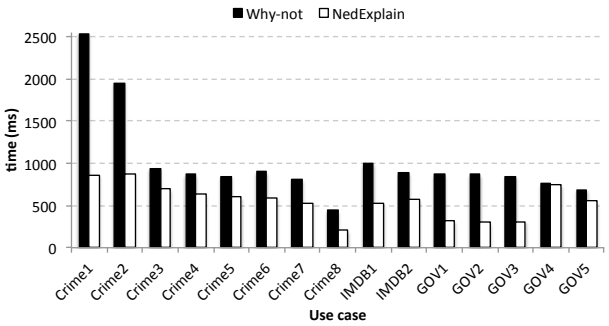


Figure 8: Why-Not and NedExplain execution time

5 Related Work

Our discussion of related work first focuses on the general context this work falls in, that is the context of data provenance and query debugging. In a second part, we focus in detail on previous work on query-based why-not provenance, with the goal of identifying the shortcomings of the previous approach.

5.1 Data provenance and Query Debugging

Recently, the problem of relational query verification has been addressed by several techniques, including data lineage [8] and data provenance [9], sub-query result inspection [10], or visualization [11]. More generally, methods for debugging declarative programming languages [12] may also apply. NedExplain can be classified as an approach to compute a special type of data provenance, referred to as why-not provenance [3, 13, 14]. Below discussion focuses on existing approaches to compute such why-not provenance.

Algorithms computing why-not provenance can be classified with respect to the output they generate. We distinguish between instance-based, query-based, and modification-based why-not provenance.

In the first case, the why-not provenance is given by a set of source data modifications that lead to the appearance of the missing-answer in the result of a query. In our example, a possible instance-based result includes the insertion of a tuple (`a5, Homer, 800BC, Harper`) into Authors. Algorithms computing instance-based why-not provenance include Missing-Answers [15] and Artemis [16]. For readers familiar with data provenance, this type of why-not provenance is analogous to why-provenance [17].

Opposed to that, query-based why-not provenance is, in a very broad sense, the counterpart of how-provenance [18]. This type of why-not provenance focuses on finding subqueries responsible for pruning the missing-answer from a query result, as illustrated in Example 1.1. We will discuss the state-of-the-art algorithm for query-based why-not provenance further below, as NedExplain also produces query-based why-not provenance.

Algorithms to compute modification-based why-not provenance [19] rewrite the given SQL query such that the missing-answer appears in the query result of the rewritten query. For instance, in our introductory example, changing the selection `B.price > 20` to `B.price >= 15` and deleting the join predicate `A.pub = B.pub` would result in the inclusion of the missing-answer (`Odyssey, 800BC`) in the query result.

5.2 The Why-Not Algorithm

The main work dealing with query based why-not provenance is provided by the Why-Not algorithm [3] which we review in detail. This algorithm focuses on workflows, and a relational query can be seen as one specific type of workflow, with relational operators being the individual components forming the workflow. Considering such a workflow and the results it produces w.r.t. a given input, a user may pose a question, concerning data missing from the result. The Why-Not algorithm identifies a set of *frontier picky manipulations* that are responsible for the exclusion of the item of interest from the result, by tracing certain source data items (tuples) through the workflow. Two alternatives are proposed for traversing the workflow: a bottom-up approach and a top-down approach. Note that in our comparative evaluation, we opted for implementing the bottom-up approach, the top-down approach not being space efficient, especially when dealing with larger data sets, as it requires all intermediate results to be stored a priori. In any case, as stated in [3] both approaches produce the same set of answers, so the quality comparison between NedExplain and Why-Not can be safely performed using one of the two approaches.

Based on a detailed analysis of Why-Not, we have identified cases, detailed below, where the solutions provided by the Why-Not algorithm are not fully satisfactory when applied to the special case of relational queries.

Inaccurate selection of unpicked data. The selection of the source data items to trace in the workflow does not take into account self-joins, possibly leading to no explanation or a wrong explanation. Furthermore, the authors choose to not trace data that contributed to some result (i.e., data in the lineage of any result tuple), which also reduces the set of why-not questions for which an explanation may be returned. *Crime7* and *Crime8* are use cases illustrating this.

Local data tracing. The source data items of interest are traced independently from each other, not considering the global picture given by the occurrence of other source data items in the trace line. In this way, inaccurate manipulations are found to be responsible (or not responsible) for the missing-answers, as use cases *Crime5* and *IMDB2* showcase.

Restriction to frontier picky manipulations. Partial results are computed by the decision to return only *frontier* picky manipulations, which strongly depend on the query tree representation. For instance, refer to use cases *Crime2* and *IMDB1*.

Insufficient answer detail. Why-Not may return more than one answer (i.e., a set of manipulations), in which case it is difficult to tell the contribution of each of the returned manipulations. Use cases *Crime3* and *GOV4* demonstrate this lack of detail.

6 Conclusions and Outlook

In this paper we have addressed the issue of answering *why-not* questions. First, we have formally defined, for the first time, the domain and concepts of *Why-Not* questions and *Why-Not* answers with respect to relational queries including projection, selection, join and union. Based on these definitions, we have provided and described NedExplain, an algorithm to produce correct Why-Not answers. As discussed and validated through experiments, NedExplain is capable of providing a more complete and correct set of answers, compared to the state-of-the-art algorithm, while being competitive in terms of runtime.

In the future, besides a more thorough experimental study of algorithm behavior, we plan to extend our algorithm to also consider aggregation and set difference. Given that lineage tracing is possible through subqueries involving aggregation and grouping (according to [8]), we believe that this goal can in principle be achieved by extending the current implementation

of NedExplain, one main question is however how to compute answers efficiently. For instance, answering a question of the form "Why are there not any publishers with average book price greater 100" requires tracing all books of all book types (all are compatible tuples), which can easily become computationally prohibitive. As for including set difference, this requires tracing data that needs to make it to the result (compatible tuples) but also data that should not make it (in order not to eliminate the compatible tuples). We further plan to study the issue of computing why-not answers such that the result is invariant w.r.t. equivalent logical query rewritings.

References

[1] M. Herschel and H. Eichelberger, "The Nautilus Analyzer: understanding and debugging data transformations," in *International Conference on Information and Knowledge Management (CIKM)*, 2012, pp. 2731–2733.

[2] M. Herschel and T. Grust, "Transformation lifecycle management with nautilus," in *VLDB Workshop on the Quality of Data (QDB)*, 2011.

[3] A. Chapman and H. V. Jagadish, "Why not?" in *International Conference on the Management of Data (SIGMOD)*, 2009, pp. 523–534.

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

[5] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa, "Data exchange: semantics and query answering," *Theoretical Computer Science*, vol. 336, no. 1, pp. 89–124, 2005.

[6] T. Imieliński and J. Witold Lipski, "Incomplete information in relational databases," *Journal of the ACM*, vol. 31, no. 4, pp. 761–791, 1984.

[7] Y. Cui and J. Widom, "Practical lineage tracing in data warehouses," in *International Conference on Data Engineering (ICDE)*, 2000, pp. 367–378.

[8] Y. Cui, J. Widom, and J. L. Wiener, "Tracing the lineage of view data in a warehousing environment," *ACM Transactions on Database Systems (TODS)*, vol. 25, no. 2, pp. 179 – 227, 2000.

[9] J. Cheney, L. Chiticariu, and W. C. Tan, "Provenance in databases: Why, how, and where," *Foundations and Trends in Databases*, vol. 1, no. 4, 2009.

[10] T. Grust and J. Rittinger, "Observing sql queries in their natural habitat (preprint)," *ACM Transactions on Database Systems*, vol. 0, no. 0, 2012.

[11] J. Danaparamita and W. Gatterbauer, "QueryViz: helping users understand SQL queries and their patterns," in *International Conference on Extending Database Technology (EDBT)*, 2011, pp. 558–561.

[12] J. Silva, "Debugging techniques for declarative languages: Profiling, program slicing and algorithmic debugging," *AI Communications*, vol. 21, no. 1, pp. 91–92, 2008.

[13] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu, "WHY SO? or WHY NO? Functional causality for explaining query answers," in *VLDB workshop on Management of Uncertain Data (MUD)*, 2010, pp. 3–17.

[14] Z. He and E. Lo, "Answering why-not questions on top-k queries," in *International Conference on Data Engineering (ICDE)*, 2012, pp. 750–761.

[15] J. Huang, T. Chen, A. Doan, and J. F. Naughton, "On the provenance of non-answers to queries over extracted data," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 1, no. 1, pp. 736–747, 2008.

[16] M. Herschel and M. A. Hernández, "Explaining missing answers to SPJUA queries," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1, pp. 185–196, 2010.

[17] P. Buneman, S. Khanna, and W. C. Tan, "Why and where: A characterization of data provenance," in *International Conference on Database Theory (ICDT)*, 2001, pp. 316–330.

[18] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *Principles of Database Systems (PODS)*, 2007, pp. 31–40.

[19] Q. T. Tran and C.-Y. Chan, "How to ConQueR why-not questions," in *International Conference on the Management of Data (SIGMOD)*, 2010, pp. 15 – 26.