Query-Based Why-Not Provenance with NedExplain
Nicole Bidoit, Melanie Herschel, Katerina Tzompanaki

To cite this version:
Nicole Bidoit, Melanie Herschel, Katerina Tzompanaki. Query-Based Why-Not Provenance with NedExplain. Extending Database Technology (EDBT), Mar 2014, Athens, Greece. hal-00962157

HAL Id: hal-00962157
https://inria.hal.science/hal-00962157
Submitted on 20 Mar 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
ABSTRACT
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 we lost data pertinent to the expected outcome. 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 NedExplain, an algorithm to explain data missing from a query result. NedExplain computes the why-not provenance for monotone relational queries with aggregation. 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.

Categories and Subject Descriptors
[Data Curation, Annotation and Provenance]: [Data Quality]

Keywords
data provenance, lineage, query analysis, data quality

1. INTRODUCTION
In designing data transformations, e.g., for data cleaning tasks, developers often face the problem that they cannot properly inspect or debug the individual steps of their transformation, 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 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 manually 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.

To improve on this manual analysis of query behavior and to ultimately help a developer in fixing the transformation, the Nautilus project [13] aims at providing semi-automatic algorithms and tools for query analysis [12], modification, and testing. This paper focuses on the analysis phase, and more specifically, proposes a novel algorithm tackling the sub-problem of explaining missing-answers. Note that explaining missing-answers is not only pertinent for query analysis and debugging, but it also applies to other domains, e.g., to what-if analysis focusing on the behavior of a query.

Explaining missing answers: running example. Very recently, approaches to explain missing-answers of relational and SQL queries have been proposed. This paper focuses on algorithms producing query-based explanations, as illustrated below.

Example 1.1. Consider the SQL query shown in Fig. 1, both in its SQL and query tree form. Ignore the operator labels $m_{Q_2}$ in the query tree for now. Let us further assume the database instance shown in Fig. 1(b). Based on these data and query, the query result includes only one tuple, i.e., (Sophocles, 49).

Assume that we now wonder why we do not find in the result a tuple with author name Homer and average price greater than 25 (assuming some knowledge on the source data), or more generally, why we do not find any other tuple with a name different from Homer or Sophocles. For this why-not question, two query-based explanations, in the form of a query tree, exist: (1) the selection on attribute dob is too strict to let any author named Homer pass (indeed, the compatible source tuple $t = (a_1, Homer, 800BC)$),
which is a candidate for contributing value Homer to the result, has dob = 800BC, so the output of the selection contains no successor of t) and (2) the join between A and AB prunes the only author with name different than Homer or Sophocles.

Related Work. Our work on query-based why-not provenance falls into the wider research area of data provenance and query debugging. We briefly review relevant works in this context.

Recently, the problem of relational query and more generally data transformation verification has been addressed by several techniques, including data lineage [5] and data provenance [3], subquery result inspection [9], or visualization [6], or query specification simplification [12, 17, 18]. More generally, methods for debugging declarative programming languages [19] may also apply.

The algorithms computing can be categorized w.r.t. the output they generate. We distinguish between instance-based, query-based, and modification-based why-not provenance.

Instance-based why-not provenance describes a set of source data modifications that lead to the appearance of the missing-answer in the result of a query. In our running example, a possible instance-based result includes the insertion of a tuple \((a1, Homer, 801BC)\) into \(A\) implying the deletion of \((a1, Homer, 800BC)\) (due to key constraints). Algorithms computing instance-based why-not provenance include Missing-Answers [15] and Artemis [14].

As opposed to that and as illustrated in our running Example 1.1, query-based why-not provenance focuses on finding sub-queries responsible for pruning the missing-answer from a query result. The state-of-the-art algorithm computing query-based why-not provenance, called Why-Not algorithm [2], is designed to compute query-based why-not provenance on workflows and also applies to relational queries when considering relational operators as the individual manipulations of the workflow, as presented in [2]. Considering such a workflow and the result it produces w.r.t. a given input, a user may specify a set of tuples missing from the result (so called missing-answers). However, as we will see throughout this paper, the Why-Not algorithm has several shortcomings that may yield incomplete or even incorrect results.

In order to compute modification-based why-not provenance, algorithms [10, 20] rewrite the given SQL query so that the missing-answer appears in the query result of the rewritten query. For instance, in our introductory example, changing the selection condition \(A.dob > 800BC\) to \(A.dob >= 800BC\) would result in the inclusion of the answer (Odyssey, 800BC) in the query result, which satisfies the user question.

Very recently, [11] has proposed an algorithm to compute hybrid why-not provenance, a combination of instance-based and query-based why-not provenance. This work is orthogonal to the work presented here, and builds on previous work [14]. The algorithm in [11] may however benefit from any query-based why-not provenance method improving the state of the art and thus from our method NedExplain.

Shortcomings of the Why-Not Algorithm. Overall, the shortcomings of [2] are linked to processing queries with self-join, empty intermediate results, the formulation of insufficiently detailed answers, and an inappropriate selection of compatible source data and their successors. We postpone a detailed discussion on these points to Sec.4, as it requires further technical details. However, let us stress two issues based on our running example.

Consider the subquery \(Q_2\) (Fig. 1(c)) of our running example. The output of \(Q_2\) consists of three tuples, e.g., one based on the join between tuples \(t_4, t_5, t_6, t_7\), and \(t_2\), denoted \(t_4 t_5 t_6 t_7 t_2\) for short. Hence, the output of \(Q_2\) is \(\{t_4 t_5 t_6 t_7 t_2, t_4 t_5 t_6 t_7, t_5 t_6 t_7\}\). Let us consider the question: Why does the output not contain a tuple with the author Homer and with a price 49? Intuitively, we see that the responsible operator is the second join between B and AB, as Homer is not associated to a book with price 49. However, in this case, the Why-Not algorithm [2] does not return any answer. The reason for this is that there are some result tuples with the name Homer \((t_4 t_5 t_2, t_5 t_6 t_7)\), and another one containing a price 49 \((t_5 t_6 t_7)\). So, ignoring that the expected values are not part of the same result tuple, the algorithm comes to the conclusion that the missing result is in fact not missing! From a technical point of view, this is due to the notion of compatible tuples adopted by the Why-Not algorithm (actually called unpicked data items in the original publication), which are input tuples that contain pieces of data of the missing answer.

Why-Not [2] may also return inaccurate results, because of the way it traces compatible tuples in the query tree. To illustrate this, let us change the subquery \(Q_3\) of our running example to \(\sigma_{A.name=1800}\) and consider the previous Why-Not question on the output of \(Q_3\), which is now empty. To answer the Why-Not question, the Why-Not algorithm [2] identifies as compatible the tuples \(t_4\) from the Authors relation and the tuple \(t_5\) from the Books relation. As we saw before, the output of \(Q_3\) contains the tuples \(t_4 t_5 t_2, t_4 t_5 t_1, t_5 t_6 t_7, t_5 t_6 t_5\). The two first tuples allow to trace \(t_4\) and the third one to trace \(t_5\). So, the Why-Not algorithm [2] identifies them as successors of the compatible tuples and will continue tracing them until it fails at \(Q_3\) because of the selection. The answer returned by [2] is \(Q_3\). However, as shown before, Homer is not associated to a book with price 49 and so the uppermost join in \(Q_3\) is also responsible for not outputting the desired result although \(Q_2\) is not returned by Why-Not [2]. From a technical point of view, this is due to a too permissive notion of successor tuple.

Contribution. The previous observations w.r.t Why-Not [2] have motivated us to investigate a novel algorithm, named NedExplain1. Our contribution is:

- **Formalization of query-based why-not provenance.** The current paper provides a formalization of query-based explanations for Why-Not questions that was missing in [2]. It relies on new notions of compatible tuples and of their valid successors. This definition subsumes the concepts informally introduced previously. It covers cases that were not properly captured in [2]. Moreover it takes into account queries involving aggregation (i.e., select-project-join-aggregate queries, or SPJQA queries for short) and unions thereof.

- **The NedExplain Algorithm.** Based on the previous formalization, the NedExplain algorithm is designed to correctly compute query-based explanations given a union of SPJQA queries, a source instance, and a specification of a missing-answer within the framework our definitions provide.

- **Comparative evaluation.** The NedExplain algorithm has been implemented for experimental validation. Our study shows that NedExplain overall outperforms Why-Not, both in terms of efficiency and in terms of explanation quality.

- **Detailed analysis of Why-Not.** We review in detail Why-Not [2] in the context of relational queries and show that it has several shortcomings leading it to return no, partial, or misleading explanations.

Organization. In Sec. 2, we set the theoretical foundation of our algorithm. In Sec. 3, we introduce and discuss NedExplain in detail. The experiments and comparative evaluation are presented in Sec. 4. Finally, we conclude in Sec. 5.

1The name is inspired by the name of one of the Nautilus’ passengers in Jules Verne’s novel 20,000 Leagues under the sea, and also stands for non-existing-data-explain.
2. QUERY-BASED EXPLANATION

We assume that the reader is familiar with the relational model [1], 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. 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 Relation 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 \( \text{type}(t) \), 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 \( \text{type}(R)=\{R.A_1, \ldots, R.A_n\} \). Note that each attribute name \( A_i \) in \( \text{type}(R) \) is qualified by the relation name \( R \).

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

Queries. As relation schema attributes are qualified, two relation schemas always have disjoint types. To define natural join and union, we thus introduce renaming.

**Definition 2.1 (Renaming \( \nu \)).** Let \( T_1 \) and \( T_2 \) be two disjoint types. A renaming \( \nu \) w.r.t. \( T_1 \) and \( T_2 \) is a set of triples \((A_1, A_2, A_{\text{new}})\) where \( A_i \in T_i \) and \( A_{\text{new}} \notin T_1 \cup T_2 \) is a new unqualified attribute. The co-domain of a renaming \( \nu \), denoted \( \text{cod}(\nu) \), is the set \( \{A_{\text{new}} \mid (A_1, A_2, A_{\text{new}}) \in \nu\} \).

Let \( T \) be a type, the mapping \( \nu(T) \) associates any \( A_i \in T \) to \( A_{\text{new}} \) if \((A_1, A_2, A_{\text{new}}) \in \nu \) and \((A_1=A_i) \lor (A_2=A_i)\), or otherwise to \( A_i \) itself.

With renaming in place, we now define the queries we consider. Essentially, we cover unions of select-project-join (SPJ) or select-project-join-aggregate (SPJQ) queries.

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

1. \([R_i]\) is a query \( Q \) with input schema \( R_i \) and target type \( \text{type}(R_i) \), \( i \in [1, n] \). \([R_i]\) has no proper subquery.

2. Let \( Q_1, Q_2 \) be queries with input schemas \( S_1, S_2 \), and target types \( \text{type}(Q_1), \text{type}(Q_2) \). Assuming \( S_1 \cap S_2 = \emptyset \).

3. Let \( Q_1 \) be a query according to (1) and (2) and \( G \subseteq \text{type}(Q_1) \). Let also \( F = \{f_1(A_1) \rightarrow A_1', \ldots, f_n(A_n) \rightarrow A_n' \} \) be a list of aggregation function calls with \( f_i \in \{\text{sum, count, avg, min, max} \} \) and \( A_i \in \text{type}(Q_1) \) that are associated with the new attribute names \( A_i' \), and let \( \text{Agg} = \{A_1', \ldots, A_n' \} \). Then, \( \alpha_G(Q_1) \) is a query \( Q \) with input schema \( S_1 \) and target type \( G \cup \text{Agg} \).

4. \([Q_1] \cup \nu \ [Q_2] \) is a query \( Q \) where \( \nu \) is a renaming w.r.t. \( \text{type}(Q_1) \) and \( \text{type}(Q_2) \) if \( \nu(\text{type}(Q_1))=\nu(\text{type}(Q_2)) \), and \( Q_1 \) and \( Q_2 \) are queries according to (1), (2), and (3).

For instance, in our example the join \( m_{\text{aid}} \) (see \( m_{Q1} \) in Fig. 1(c)) stands for \( m_{\text{aid}} \). The renaming \( \nu \) in this case is the triple \((A.\text{aid}, B.A.\text{aid}, \text{aid})\) which maps the qualified attributes \( A.\text{aid} \) and \( B.A.\text{aid} \) to the new attribute \( \text{aid} \).

To simplify our discussion, we assume that subqueries are named and that two subqueries \( Q_1 \) and \( Q_2 \) have distinct target attributes. Of course, users can write their queries in a less restrictive way using traditional SQL syntax, which are then automatically translated to our query form.

Given a unary (binary) query \( Q \) built from queries \( Q_1 \) and \( Q_2 \), we consider as \( Q \)’s subqueries \( Q_1 \) (and \( Q_2 \)) as well as their respective subqueries. That is, using a standard tree representation of queries, one node corresponds to a subquery.

We further define an input instance for a query \( Q \) whose input schema is \( S_Q \), as an instance over \( S_Q \). This definition, given below, introduces \( \eta_Q \) to correctly deal with self-joins.

**Definition 2.3 (Query over a database).** A query over a database schema \( S \) is a pair \((Q, \eta_Q)\) where \( Q \) is a query with input schema \( S_Q \) and \( \eta_Q \) is a mapping from \( S_Q \) to \( S \) such that for any \( R \in S_Q \), \( R.A \in \text{type}(R) \) iff \( \eta_Q(R.A) \in \text{type}(\eta_Q(R)) \).

Given an instance \( I \) over \( S_Q \), the evaluation of \((Q, \eta_Q) \) over \( I \) is defined as the evaluation of \( Q \) over the input instance \( I_Q \) over \( S_Q \) defined by: for any \( S \in S_Q \), \( I_Q|_S = I[R] \) if \( \eta_Q(S) = R \).

Given the data and query as formalized above, we now formalize how Why-Not questions are expressed.

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. Such a predicate is a disjunction of conjunctive tuples, which are essentially attribute-value/variable pairs possibly constrained by conjunctive predicates. We start by introducing tuples with variables and then conditional tuples.

**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 \( e_i \in V \cup \text{dom}(A_i) \) for \( i \in [1, n] \) and \( \text{dom}(A_i) \) denoting the active domain of \( A_i \).

The variables of a v-tuple are similar in spirit to labeled nulls, used for instance in the context of data exchange [7]. Intuitively, the semantics associated to such variables 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, it yet should satisfy some constraints. For this reason, we resort to conditional tuples (or c-tuples for short), previously introduced for incomplete databases [16].

**Definition 2.5 (Conditional Tuple (c-Tuple)).** Let \( t_c \) be a v-tuple and let \( X_c \) be the set of variables in \( t_c \). A c-tuple \( t_c \) is a pair \((t_c, \text{cond})\) where \( \text{cond} = \bigwedge_{i=1}^n \text{pred}_i \), and for \( 1 \leq i \leq n \)

\[ \text{pred}_i \ni \text{true} \mid x_i \ni \text{cop}_x \mid x_i \ni \text{cop} \quad x_i \ni \text{variable in} \ X_c, \ a \in \text{dom}(\text{type}(x_i)) \text{, and } \text{cop} \text{ is a comparison operator } (\neq, =, <, >, \geq, \leq) \]
The type of a c-tuple \((t_e, \text{cond})\) is the type of \(t_e\). We now are ready to define what is a Why-Not question.

**Definition 2.6 (Why-Not Question).** A Why-Not question w.r.t. a query \(Q\) is a predicate \(P\) over \(Q\)’s target type \(T_Q\), where \(P = \bigvee_{i=1}^n t_i\), with \(t_i\) being a c-tuple s.t. type\(type(t_i)\) \(\subseteq T_Q\).

**Example 2.1.** The Why-Not question expressed in Eq. 1.1 corresponds to the predicate \(P = ((A \text{.name} \bowtie \text{Homer, } \text{ap} \bowtie x_1), x_1 > 25) \vee ((A \text{.name} = \bowtie x_2, x_2 \not= \text{Homer} \land x_2 \not= \text{Sophocles})).

In the sequel, we will omit the condition when it is \(\text{true}\), i.e., we may rewrite the c-tuple \((t, \text{true})\) as \(t\). Also, we consider only conditions that compare variables with constants or with variables that are local to the same relation.

As a reminder, given a query \(Q\) whose input schema is \(S_Q\), new attributes may have been introduced through join or union specifications. These new attributes are well identified and linked to the attributes may have been introduced through join or union specifications that compare variables with constants or with variables that are local to the same relation.

**Definition 2.7 (Unrenamed Predicate w.r.t. a Query Q).** Let \(t_e\) be a c-tuple and \(\nu\) a renaming. Given any \((A_1, A_2, A_{\text{new}}) \in \nu\), if \(A_{\text{new}} \in \text{type}\(t_e)\), we replace each \(A_{\text{new}}\) in \(t_e\) by \(A_1\), denoted as \(\nu[1](t_e)\). We proceed analogously for \(A_2\), yielding \(\nu[2](t_e)\).

Let \(Q\) be a query. The mapping \(\text{UnR}_{\nu}Q\) associates to \(t_e\) a predicate defined by:

1. If \(Q = [R_1]\) then \(\text{UnR}_{\nu}Q(t_e) = t_e\).
2. Let \(Q_1, Q_2\) be queries:
   - if \(Q = [Q_1] \bowtie [Q_2]\), then
     \(\text{UnR}_{\nu}Q(t_e) = \text{UnR}_{\nu}Q_1(\nu[1](t_e)) \bowtie \text{UnR}_{\nu}Q_2(\nu[2](t_e))\)
   - if \(Q = [Q_1] \cup [Q_2]\), then
     \(\text{UnR}_{\nu}Q(t_e) = \text{UnR}_{\nu}Q_1(\nu[1](t_e)) \cup \text{UnR}_{\nu}Q_2(\nu[2](t_e))\)
   - if \(Q = \pi_W[Q_1]\), \(Q = \sigma_{C_1}(Q_1)\), or \(Q = \sigma_{C_2}[Q_1]\) then
     \(\text{UnR}_{\nu}Q(t_e) = \pi_W[\text{UnR}_{\nu}Q_1](t_e)\).

If \(P\) is the predicate \(\bigwedge_{i=1}^n t_i\), then the unnamed predicate associated with \(P\) given the query \(Q\) is \(\bigwedge_{i=1}^n \text{UnR}_{\nu}Q(t_i)\).

**Example 2.2.** Assume that our sample query \(Q\) includes one more output attribute, i.e., \(T_Q = \{A \text{.name}, \text{aid}, \text{ap}\}\), and assume the renaming \(\nu = \{(A \text{.aid}, \text{A.aid}), \text{A.aid}\}\).

For the predicate \(P = (A \text{.name} \bowtie \text{Homer, A.aid} \bowtie \text{aid1}, \text{ap} \bowtie x_1)\), the attribute \(\text{A.aid}\) can be renamed to \(\text{A.aid}\) and to \(\text{A.aid}\), two qualified attributes that cannot be further unnamed. So, the unnamed predicate \(P\) is \((A \text{.name} \bowtie \text{Homer, \text{A.aid}1}, \text{ap} \bowtie x_1)\) as \((A \text{.name} \bowtie \text{Homer, \text{A.aid}1}, \text{ap} \bowtie x_1)\).
Definition 2.9 (Tuple successor w.r.t. a query).
Let \( Q \) be a query over \( S_Q \) and \( I \) be an instance over \( S_Q \). A tuple \( t \in Q(I) \) is a successor of some \( t_2 \in D \) w.r.t. \( Q \) if for \( Q = UOp[Q_1] \) (resp. \( Q = |Q_1|BOp(Q_2) \), there exists some \( t' \in Q_1(I_1) \) (resp. \( t' \in Q_1(I_1) \cup Q_2(I_2) \)) such that \( Q \) is a successor of \( t' \) w.r.t. \( m_Q \) and \( Q_1(I_1) \) (resp. \( Q_1(I_1) \cup Q_2(I_2) \)) and either \( t' = t_2 \) or \( t' \) is a successor of \( t_2 \) w.r.t. \( Q_1 \) (resp. \( Q_1 \) or \( Q_2 \)). Here, \( I \) is the instance over \( S_Q \), defined by \( I = I \setminus \{ s \} \) for \( i = 1, 2 \).

Next, \( V S(Q, I, D, t) \) denotes, for a given instance \( I \), the set of valid successors of \( t $$\in$$ D \) w.r.t. \( Q \).

In our running example, consider the subquery \( Q_2 \) on the input instance \( I \) shown in Fig. 1(b). Then let \( D = \{ t_4, t_2 \} \cup I_{AB} \) and consider the tuple \( t_4 $$\in$$ D \). The output of \( Q_2 \) is \( Q_2(I) = \{ t_4, t_2, t_3, t_5 \} \). Each output tuple is represented by the identifiers of the tuples in its lineage. We say that the output tuple \( t_4, t_2 \) is a valid successor of \( t_4 \) because it is a successor of \( t_4 \) w.r.t. \( Q_2(I) \) and \( I \) and its lineage is included in \( D \) (i.e., \( t_4, t_2, t_3, t_5 \) are all in \( D \)). On the contrary, the output tuple \( t_5 \) is not a valid successor of \( t_4 \) even though it is a successor of \( t_4 \), because the tuple \( t_4 \) which is in the lineage of \( t_5 \), is not in \( D \).

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

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_2 $$\in$$ D \). The definitions, given below, are illustrated in Fig. 3.

Definition 2.10 (Picky manipulation). Let \( m \) be a manipulation, \( I \) be a well typed input instance for \( m \) and \( D $$\subseteq$$ I \). Then \( m \) is a picky manipulation w.r.t. \( D \) and \( t_2 $$\in$$ D \), if there is no valid successor \( t \) of \( t_2 \) in \( I \).

Definition 2.11 (Picky query). Let \( Q \) be a query over \( S_Q \) and \( I \) an input instance for \( Q \) and \( D $$\subseteq$$ I \) a set of tuples. Let \( t_2 \) be a tuple in \( D \).
Assuming that \( Q = |Q_1|BOp(Q_2) \) and that \( t_2 $$\in$$ I \) (the case of \( t_2 $$\in$$ I \) is dual), \( Q \) is picky w.r.t. \( D \) and \( t_2 \) if
3. THE NEDEXPLAIN ALGORITHM

Based on the framework our definitions provide, we now present NeExplain, an algorithm that takes as input a predicate \( P \) over the output type \( \text{type}(Q) \) of a query \( Q \) over a database schema \( S_Q \) and the query input database instance \( I_Q \). We limit \( Q \) to a union of SPJ queries, deferring the addition of further operators to future work. NeExplain supports the computation of any type of Why-Not answer (see Sec. 2.5). However, due to the limited space, we focus our discussion on outputting detailed Why-Not answers.

3.1 Preprocessing

NeExplain starts with a preprocessing phase, consisting of the steps described below:

1) Unrenaming. First, we rename \( P \) as defined in Def. 2.7. Thereby, we obtain \( \text{unr}(P) = \bigcup_{i=1}^{m} t_i \), where every \( t_i \) contains only qualified attributes w.r.t. \( I_Q \) or aggregated attributes. This step is performed only once.

We continue by performing the following procedures regarding one \( t_c \) of \( \text{unr}(P) \) at a time, as indicated by the first two lines of Alg. 1. The union of the results produced for each \( t_c \) corresponds to the final Why-Not answer w.r.t. \( P \).

2a) CompatibleFinder. From \( t_c \), we can easily compute the direct compatible set of tuples \( I_{\text{Dir}} \) w.r.t. \( t_c \) by performing appropriate SELECT statements that retrieve ids\(^2\) 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 source tuple, also illustrated below. In parallel, \( I_{\text{Indir}} \) is determined, as defined in the context of Def. 2.8.

2b) Canonicalize \( Q \). A relational query \( Q \) may result in various equivalent query plans (trees) and similarly to [2, 5], we choose a canonical query tree representation that limits the equivalent query trees to consider. The following two rationales guide our choice of canonical query tree representation that differs from the canonical tree representation of [2].

First, we favor finding selections as Why-Not answers over finding joins, as selections are easier to inspect and change by a developer. Furthermore, this choice allows us to potentially reduce the runtime of NeExplain, since it allows us to push down selections (and as we shall see, we traverse and evaluate operators of the query tree in a bottom-up order).

Second, as described by Def. 2.12, we need to determine if a subquery (tree node) is picky and whether the condition \( t_c, \text{cond}_a \) is satisfied by the subquery’s input and not in its output. In order to maximize the number of subqueries for which we can verify these conditions, we organize joins such that we obtain a view \( V \) of minimal query size where \( \text{type}(V) \supseteq G \cup \{ A_1, \ldots, A_n \} \) and no cross product is necessary. Intuitively, \( V \) corresponds to the subquery closest to the leaf level in the query tree joining all grouped and aggregated attributes. We refer to \( V \) as breakout subquery. Obviously, for queries without aggregation, the condition \( \text{type}(V) \supseteq G \cup \{ A_1, \ldots, A_n \} \) is trivially satisfied for any leaf node, i.e., for any \( V \in \text{Inst}_Q \) (as \( G \cup \{ A_1, \ldots, A_n \} = \emptyset \)), which results in all leaf nodes being breakpoint queries. Similarly, all leaf nodes representing relations in \( I_Q \setminus I_V \) can be considered as breakpoint queries. We refer to the set of all breakpoint queries, i.e., \( V \cup (I_Q \setminus I_V) \) as visibility-frontier. Given the query tree with

\footnote{These queries assume that each table has a key attribute to uniquely identify a tuple. The queries can however be trivially modified to SELECT * queries if no such key exists, processing would however take more space.}

In the next definitions, we assume that \( Q \) is a query over \( S_Q \) and \( I \) is an input instance for \( Q \). We also assume that \( t_c \) is an unrenamed conditional tuple, as stated before.

Let us start by defining the detailed answer of a Why-Not question, which records: (1) the picky query per compatible tuple (if any) and (2) in the case of aggregation, the subquery violating the conditions on aggregated values.

**Definition 2.12 (Detailed Why-Not Answer).** The detailed Why-Not answer of \( t_c \) w.r.t. \( Q \) and \( I \), denoted \( \text{dW}_Q(t_c) \), is defined as follows given that \( Q' \) below is a unary query of the form \( \bigcup Q' \) (resp. a binary query of the form \( \{Q_1,BOP(Q_2)\} \):

\[
\bigcup_{t_x \in \text{Dir}} \{ (t_x, Q') | Q' \text{ subquery of } Q \text{ and } Q' \text{ picky w.r.t. } \text{Dir}^{t_x} \cup \text{Indir}^{t_x} \text{ and } t_x \}
\]

\[
\bigcup \{ (\perp, Q') | V \text{ proper subquery of } Q' \text{ and } Q'_I(\perp) \text{ resp. } Q'_I(\perp) \cup Q'_I(\perp) \neq t_c, \text{cond}_a \text{ and } Q' \notin t_c, \text{cond}_a \}
\]

The second part of this definition ensures that the conditions on aggregated values are verified on the input of \( Q' \), but not on its output.

**Example 2.6. In our running example \( V = Q' \). The detailed Why-Not answer for the first part of our Why-Not question (i.e., \( t_c = ((\text{Homer}, x_1), x_1 > 25) \)) is \( \{ (t_4, Q_3) \} \) which is QW picky w.r.t. \( t_4 \) and \( \{ t_4 \} \cup \{ (Q_A, B) \cup (Q_B) \} \) and the data provided by \( V \) may satisfy the aggregation condition (e.g., applying the aggregation on the tuples present in \( V \) yields an average price of 30, which is above 25), whereas the empty output of \( Q_2 \) does not satisfy this condition.

In general, this detailed answer may be too overwhelming for a user (due to the potentially large number of picked compatible tuples). Thus, we also define a condensed Why-Not answer that only provides the set of picky subqueries to the user, e.g., \( \{ Q_3 \} \) in the previous example.

**Definition 2.13 (Condensed Why-Not Answer).** The condensed Why-Not answer for \( t_c \) w.r.t. \( Q \) and \( I \) is defined as \( \text{cW}_Q(t_c) \) = \( \{ Q' | (d_x, Q') \in \text{dW}_Q(t_c) \} \).

Finally, we also define a secondary Why-Not answer that considers the indirect compatible set \( \text{Indir}^{t_x} \). Recall that \( \text{Indir}^{t_x} \) includes data necessary to produce \( t_x \), but that is not constrained by \( t_c \) (i.e., its necessity is only imposed by \( Q \)). Consequently, missing \( t_x \) may also be caused by the “disappearance” of \( \text{Indir}^{t_x} \), a case we capture with the secondary Why-Not answer.

**Definition 2.7. Assume we replace the right child of \( Q_2 \), i.e., \( B \), with the subquery \( Q'_1 \). Then, \( Q_1 = \bigcup \{ Q' | (d_x, Q') \in \text{cW}_Q(t_c) \} \).**

As a reminder, \( S^{t_x} \) is the set of relation schemas typing the tuples in \( \text{Dir}^{t_x} \) and thus \( S_0 = S^{t_x} \) is the set of relation schemas typing the tuples in \( \text{Indir}^{t_x} \).

**Definition 2.14 (Secondary Why-Not Answer).** Let \( S \in S_Q \setminus S^{t_x} \). We denote by \( Q_S \) the subquery of \( Q \) s.t. \( Q_S \) is picky w.r.t. \( I \) and some \( d \in I_S \), and for any \( d' \in I_S \), there is no successor of \( d' \) w.r.t. \( Q_S \). Then, the secondary Why-Not answer of \( t_c \) w.r.t. \( Q \) and \( I \) is \( \text{sW}_Q(t_c) = \{ Q_S | S \in S_Q \setminus S^{t_x} \} \).

Fig. 3(c) illustrates the secondary Why-Not answer.
minimized breakpoint queries, we place the selections above and closest to the visibility-frontier to satisfy our first rationale.

Ex. 3.1 clarifies how we obtained the canonical query tree illustrated in Fig. 1 (c). Based on the appropriate selection of the visibility frontier at $Q_2$, the selection on $A.dob$ has been placed as close to this frontier as possible.

In the sequel, we denote our canonical query tree satisfying the above rationales as $T$.

2c) Primary global structure $\text{Tab}_Q$. NedExplain relies heavily on one main global structure, denoted $\text{Tab}_Q$. $\text{Tab}_Q$ stores intermediate computations as well, as discussed later. More specifically, $\text{Tab}_Q$ contains the following labeled entries for each subquery $m$ of $Q$.

- **Input**: the input tuple set of subquery $m$
- **Output**: the output tuple set of subquery $m$
- **Compatibles**: the set of tuples defined by
$$\{t_i \mid t_i \in m.\text{Input} \land (t_i \in \text{Dir}^e \lor \text{succ} \in \text{m.Input} \land \text{Select}^e \land \text{w.r.t.} \text{m})\}$$
- **Level**: the depth of $m$ in $T$ (the root having level 0)
- **Parent**: the parent node (subquery) of $m$ in $T$
- **Op**: the root operation of $m$
- **Dir**: the set of tuples defined by
$$\{t \mid t \in m.\text{Input} \land (t \in \text{Dir}^e \lor \text{succ} \in \text{m.Input} \land \text{Select}^e \land \text{w.r.t.} \text{m})\}$$

To refer to the entry labeled $l$ of a subquery $m$, we write $m.l$, e.g., $m.\text{level}$ refers to the level of subquery $m$.

Initialization is trivial for $m.\text{Op}$, $m.\text{Parent}$ and $m.\text{Level}$ based on $T$. For $m.\text{Input}$, initialization is possible for any $m$ that is a base relation, based on $m.\text{Input}=\{I_{Q_i}\}$ where $m=\{R_i\}$, $R_i \in S_Q$. Then, we initialize $m.\text{Compatibles}$ for any $m$ that is a base relation by $m.\text{Compatibles}=(\text{Dir}^e|_{R_i})$ where $m=\{R_i\}$, $R_i \in S_Q$.

The rest of the entries get updated during the execution of the algorithm. In order to efficiently access the information in $\text{Tab}_Q$ that is necessary during processing, subqueries are stored in order of decreasing depth ($m.\text{Level}$) in the query tree. We access subquery $m$ at position $i$ using the notation $m=\text{Tab}_Q[i]$.

2d) Secondary global structures. Apart from $\text{Tab}_Q$, we make use of some other global structures, which are:

- **EmptyOutputMan**: the set of subqueries producing the empty set, used to determine the secondary Why-Not answer.
- **Non-PickyMan**: the set subqueries, producing successors of compatible tuples.
- **PickyMan**: the set of pairs $(m, \text{blocked})$, where $m$ is a subquery and $\text{blocked}=\{t | t \in m.\text{Input} \land m \text{ is picky w.r.t.} t \text{ and } \text{Dir}^e \cup \text{IN} \text{Dir}^e \text{ for } t\}$.

This structure allows us to determine both the detailed and the condensed Why-Not answer.

**Example 3.1.** Given our running example and the c-tuple $t_c=((A.\text{name}=\text{Homer}, \text{ap}2x), \text{r}1>25)$, $\text{CompatFinder}$ executes the SQL query `SELECT A.\text{aid} FROM A WHERE A.\text{name} = \text{Homee}` to obtain $\text{Dir}^e = \{t_4\}$. Canonicalization of the query in Fig. 1(a) results in the query tree of Fig. 1(c), where the minimum subquery containing both $A.\text{name}$ and $B.\text{price}$ is the subquery $Q_2$, i.e., $V = \{A \,(A.\text{aid}, AB.\text{aid}, \text{aid})\ AB\, B.\text{bid}, B.\text{bid}, B.\text{bid}\}$. Here, $V_x = I_A \cup I_AB \cup I_B$ and $I_O \setminus I_V = \emptyset$, so our visibility-frontier consists of $V$ only. The selection operator $\sigma_{A.\text{name}=\text{Horse}, B.\text{price}>1000}$ is then placed just above $V$ (i.e., $Q_2$). Tab. 1 shows the initialization of $\text{Tab}_Q$ given the canonical query tree of Fig. 1(c).

<table>
<thead>
<tr>
<th>Entry label</th>
<th>A</th>
<th>AB</th>
<th>$m_{Q_1}$</th>
<th>B</th>
<th>$m_{Q_2}$</th>
<th>$m_Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$I_{A}, I_{AB}$</td>
<td>$I_B$</td>
<td>$I_{Q_2}$</td>
<td>$I_{Q_3}$</td>
<td>$I_{Q_3}$</td>
<td></td>
</tr>
<tr>
<td>Comptables</td>
<td>${t_4}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>Level</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Parent</td>
<td>$m_{Q_1}$</td>
<td>$m_{Q_2}$</td>
<td>$m_{Q_2}$</td>
<td>$m_{Q_2}$</td>
<td>$m_{Q_2}$</td>
<td></td>
</tr>
</tbody>
</table>

| Op          | relation schema | relation schema | $\emptyset$ | relation schema | $\emptyset$ | $\sigma$ | $\alpha$ |

### Table 1: Primary global structure $\text{Tab}_Q$

#### Algorithm 1: $\text{NedExplain}$

1. **CompatibleFinder**;
2. **Canonicalize(Q)**;
3. Initialization of global structures: $\text{Tab}_Q$, Non-PickyMan, EmptyOutputMan, PickyMan;
4. for $(i=0, \ldots, \text{Tab}_Q.\text{size}() - 1)$ do
5. \hspace{1cm} if $(\text{checkEarlyTermination}(m))$ then
6. \hspace{1cm} \hspace{1cm} return $\text{DetailedAnswer}$;
7. \hspace{1cm} m.Output $\leftarrow$ m.Input;
8. \hspace{1cm} p$\leftarrow m.\text{Parent}$;
9. \hspace{1cm} p.Input $\leftarrow$ p.Output $\cup$ m.Output;
10. \hspace{1cm} if $(m.\text{Output} = \emptyset)$ then
11. \hspace{1cm} \hspace{1cm} EmptyOutputMan $\leftarrow$ EmptyOutputMan $\cup \{m\}$;
12. \hspace{1cm} \hspace{1cm} if $(m.\text{Compatibles} = \emptyset)$ then
13. \hspace{1cm} \hspace{1cm} \hspace{1cm} PickyMan $\leftarrow$ PickyMan $\cup \{m, m.\text{Compatibles}\}$;
14. \hspace{1cm} \hspace{1cm} else
15. \hspace{1cm} \hspace{1cm} \hspace{1cm} if $(m.\text{Op} = \text{relation schema})$ then
16. \hspace{1cm} \hspace{1cm} \hspace{1cm} p.Compatibles $\leftarrow$ p.Compatibles $\cup m.\text{Compatibles}$;
17. \hspace{1cm} \hspace{1cm} else
18. \hspace{1cm} \hspace{1cm} \hspace{1cm} p.Compatibles $\leftarrow$ p.Compatibles $\cup m.\text{Compatibles}$;
19. \hspace{1cm} \hspace{1cm} Non-PickyMan $\leftarrow$ Non-PickyMan $\cup \{m\}$;
20. \hspace{1cm} end
21. return null;

#### 3.2 Computing Why-Not Answers

After initialization, we visit the subqueries of $Q$ in the order given by $\text{Tab}_Q$, which corresponds to the decreasing depth of subqueries in the query tree. At each step, we identify the successors of the compatible tuples and keep track of the picky and non-picky subqueries along the way. In the end, we return the Why-Not answer (due to the lack of space, we focus only on the detailed Why-Not answer here).

Alg. 1 is the main algorithm of $\text{NedExplain}$. It takes as input one unrenamed c-tuple $t_c$, so it will be executed once for each $t_c$ in $\text{UnR(P)}$. The overall answer w.r.t. $P$, will then be the union of all answers for each $t_c$.

Alg. 1 initializes the compatible tuple sets as well as the global structures w.r.t. $t_c$ in lines 1–3. Next, it iterates through all the subqueries stored in $\text{Tab}_Q$ until $\text{checkEarlyTermination}$ (Alg. 2) called at line 6 returns true or the last entry in $\text{Tab}_Q$ has been reached. If $\text{checkEarlyTermination}$ returns true, we compute and return the detailed Why-Not answer. Otherwise, we continue with the evaluation of the current subquery $m$ on its input (line 8) and maintain the entries of $m$’s parent $p$ in $\text{Tab}_Q$. We also maintain the secondary global structures EmptyOutputMan and PickyMan (lines 11–14). For all subqueries except for those that correspond to relation schemas (as identified by $m.\text{Op}$), $\text{FindSuccessors}$ (Alg. 3) finds possible successors of compatible tuples in the output of the current subquery and maintains the secondary global structures PickyMan and Non-PickyMan (line 16). Otherwise, $p.\text{Compatibles}$ and the global structures update as described in lines 17–20. We now further discuss the sub-algorithms that Alg. 1 calls.

**Check for early termination (Alg. 2).** This step decides whether
Algorithm 2: checkEarlyTermination

Input: \( m \), a subquery
Output: a boolean value
1. \( i \leftarrow \) position of \( m \) in \( \text{Tab}_Q \);
2. if \((i \neq 0 \land m.\text{Level} \neq \text{Tab}_Q(i-1).\text{Level}) \) then
3. \( j \leftarrow i-1; \)
4. while \((i \geq 0 \land \text{Tab}_Q[i].\text{Level} = \text{Tab}_Q(i-1).\text{Level}) \) do
5. if \((\text{Tab}_Q[i].\text{Input} \cap m.\text{Compatibles} \neq \emptyset) \) then
6. return FALSE;
7. \( j \leftarrow j+1; \)
8. while \((i < \text{Tab}_Q.\text{size}()) \) do
9. if \((\text{Tab}_Q[i].\text{Input} \cap m.\text{Compatibles} \neq \emptyset) \) then
10. return FALSE;
11. \( i \leftarrow i+1; \)
12. else
13. return FALSE;
14. return TRUE;

Algorithm 3: FindSuccessors

Input: \( m \), a subquery
Output: the successors of tuples in \( m.\text{Compatibles} \), in the output of \( m \)
1. compOrigins \( \leftarrow \{m'[m'.\text{Input} \cap m.\text{Compatibles} \neq \emptyset \} \land (m'.\text{parent} = m)\}; \)
2. successors \( \leftarrow \emptyset; \)
3. foreach \( o \in m.\text{Output} \) do
4. if \((\text{lineage}(o) \subseteq \text{Dir}^e \cup \text{InDir}^e) \) then
5. \( \text{successors} \leftarrow \text{successors} \cup (\text{lineage}(o) \cap \text{Dir}^e); \)
6. Blocked \( \leftarrow m.\text{Compatibles} \setminus \text{successors}; \)
7. if \((\text{successors} \neq \emptyset) \) then
8. \( \text{successors} \leftarrow \text{Non-PickyMan} \cup \{m\}; \)
9. if \((\text{Blocked} \neq \emptyset \land \forall v \text{ not subquery of } m) \) then
10. \( (V \text{ subquery of } m \land \forall q \in \text{m.Input} \Rightarrow t_c, \text{cond}_a) \)
11. if \((m.\text{Output} \neq \emptyset \lor m.\text{Input} \neq \emptyset) \) then
12. \( \text{PickyMan} \leftarrow \text{PickyMan} \cup \{\text{mBlocked}\}; \)
13. return \( \text{successors} \);
In the future, we will investigate algorithms to be invariant w.r.t. equivalent query tree rewritings.

4. EXPERIMENTS

In this section we display a comparative evaluation of our algorithm with respect to the Why-Not algorithm [2]. Briefly, the Why-Not algorithm [2] identifies a set of frontier picky manipulations that are responsible for the exclusion of missing-answers from the result by tracing unpicked data items (tuples) through the workflow. Two alternatives are proposed for traversing the workflow: a bottom-up approach and a top-down approach. The main difference between the two approaches lies in the efficiency of the algorithms (depending on the query and the Why-Not question). In [2], it is stated that both approaches are equivalent as they produce the same set of answers.

We have implemented NedExplain and Why-Not (actually, its bottom-up version as it most resembles the approach of NedExplain) using Java, based on source code kindly provided by the authors of Why-Not. The original Why-Not implementation, as well as ours, relies on the lineage tracing provided by Trio (http://infobank.stanford.edu/trio/). We ran the experiments on an Oracle Virtual Machine running Windows 7 and using 2GB main of 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.

4.1 Use Cases

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). Each use case consists of a query further defined in Tab. 3 and a Why-Not question in form of a predicate \( \mathcal{P} \) as defined by Def. 2.6. The queries have been designed to include simple (Q4, Q6) and more complicated (Q1, Q3, Q5, Q7) queries, queries containing self-joins (Q3, Q4), queries having intermediate results (Q2, Q8) and SPJ queries (Q8, Q9) and SPJU queries (Q12). To pinpoint the differences between the two algorithms, some use cases consider the same query with a different predicate.

Based on these use cases, we evaluate NedExplain and Why-Not, both in terms of answer quality and efficiency.

4.2 Answer Quality

Tab. 5 summarizes the why-not answers obtained by running our scenarios on Why-Not and NedExplain. For NedExplain, we distinguish among the detailed, the condensed and the secondary Why-Not answer, as defined by Defs. 2.12–2.14.

At first sight, the answers provided by Why-Not are simpler and clearer; they generally consist of a small number of subqueries.

3For easy of presentation, in presence of renaming, we display only the new attributes introduced by renaming.

<table>
<thead>
<tr>
<th>Query</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>( \pi_{P\text{.name},C\text{.type}}({W\text{.sector}, W\text{.witnessName}, S\text{.hair,clothes}}) )</td>
</tr>
<tr>
<td>Q2</td>
<td>( \pi_{P\text{.name},C\text{.type}}(({W\text{.sector} \geq 90} \land {W\text{.witnessName} \geq 5} \land {S\text{.hair,clothes}})) )</td>
</tr>
<tr>
<td>Q3</td>
<td>( \pi_{W\text{.name},C\text{.type}}({W\text{.sector} &lt; 2 \land C\text{.sector} &gt; 5 \land C\text{.type} = \text{Aiding}}) )</td>
</tr>
<tr>
<td>Q4</td>
<td>( \pi_{P\text{.name}}(({P\text{.name}} \land {P\text{.name} \geq 2})) )</td>
</tr>
<tr>
<td>Q5</td>
<td>( \pi_{C\text{.sector}, C\text{.type}}({C\text{.sector} &gt; 5, C\text{.type} \geq 1}) )</td>
</tr>
<tr>
<td>Q6</td>
<td>( \pi_{C\text{.sector}, C\text{.type}}({C\text{.sector} \geq 10}) )</td>
</tr>
<tr>
<td>Q7</td>
<td>( \pi_{P\text{.name},C\text{.type}}({P\text{.name} \geq 2}) )</td>
</tr>
<tr>
<td>Q8</td>
<td>( \pi_{P\text{.name}}({P\text{.name} \geq 2}) )</td>
</tr>
<tr>
<td>Q9</td>
<td>( \pi_{P\text{.name}}({P\text{.name} \geq 2}) )</td>
</tr>
</tbody>
</table>

Table 3: Use cases relational queries

<table>
<thead>
<tr>
<th>Use case</th>
<th>Query</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crime1</td>
<td>Q1</td>
<td>(P\text{.name}:Hank,C\text{.type}:Car theft)</td>
</tr>
<tr>
<td>Crime2</td>
<td>Q1</td>
<td>(P\text{.name}:Roger,C\text{.type}:Car theft)</td>
</tr>
<tr>
<td>Crime3</td>
<td>Q2</td>
<td>(P\text{.name}:Roger,C\text{.type}:Car theft)</td>
</tr>
<tr>
<td>Crime4</td>
<td>Q2</td>
<td>(P\text{.name}:Roger,C\text{.type}:Car theft)</td>
</tr>
<tr>
<td>Crime5</td>
<td>Q2</td>
<td>(P\text{.name}:Hank)</td>
</tr>
<tr>
<td>Crime6</td>
<td>Q3</td>
<td>(C\text{.type}:Kidnapping)</td>
</tr>
<tr>
<td>Crime7</td>
<td>Q3</td>
<td>(W\text{.name}:Susan,C\text{.type}:Kidnapping)</td>
</tr>
<tr>
<td>Crime8</td>
<td>Q4</td>
<td>(P\text{.name}:Audrey)</td>
</tr>
<tr>
<td>Crime9</td>
<td>Q8</td>
<td>(P\text{.name}:Betsy)</td>
</tr>
<tr>
<td>Crime10</td>
<td>Q8</td>
<td>(P\text{.name}:Roger)</td>
</tr>
<tr>
<td>Imdb1</td>
<td>Q5</td>
<td>(name:Avalar)</td>
</tr>
<tr>
<td>Imdb2</td>
<td>Q5</td>
<td>(name:Christmas Story, L\text{.locationId}:USANewYork)</td>
</tr>
<tr>
<td>Gov1</td>
<td>Q6</td>
<td>(C\text{.firstname}:Christopher)</td>
</tr>
<tr>
<td>Gov2</td>
<td>Q6</td>
<td>(C\text{.firstname}:Christopher, C\text{.lastname}:MURPHY)</td>
</tr>
<tr>
<td>Gov3</td>
<td>Q6</td>
<td>(C\text{.firstname}:Christopher, C\text{.lastname}:GIBSON)</td>
</tr>
<tr>
<td>Gov4</td>
<td>Q7</td>
<td>(sponsorId:467)</td>
</tr>
<tr>
<td>Gov5</td>
<td>Q7</td>
<td>(SP\text{.sponsorId}:Lugar, E\text{.camount}:x, x, &gt;=1000)</td>
</tr>
<tr>
<td>Gov6</td>
<td>Q9</td>
<td>(name:Bennett, A\text{.state}:NY)</td>
</tr>
<tr>
<td>Gov7</td>
<td>Q12</td>
<td>(name:JOHN)</td>
</tr>
</tbody>
</table>

Table 4: Use cases

On the other hand, NedExplain provides an answer more complex in structure, but more informative. Notice that the condensed answer resembles the answer returned by Why-Not and provides an “easily-consumable” answer as well. Still, they are not the same, as we will describe later in this section. The subsequent discussion first highlights the differences of the various kinds of NedExplain answers before we compare the results of NedExplain with those that Why-Not produces. Due to the lack of space, we limit the discussion to representative use cases.

Detailed vs. condensed and secondary Why-Not answers. Consider the Crime6 use case associated to the query tree for Q3 depicted in Fig. 4(b). The condensed answer (but not the Why-Not algorithm answer, as described later) indicates that m1 is a picky subquery. The detailed answer consists of pairs of the form \((\alpha_{i}, \beta_{i}), i = 1, \ldots, 11\). For this case, the detailed answer does not provide substantial new insights compared to the condensed answer, as m1 is responsible for pruning all compatible tuples. In this situation, the condensed answer is the most appropriate to return to the user. However, in other cases, the simplicity of the condensed answer
Table 5: Why-Not and NedExplain answers, per use case

<table>
<thead>
<tr>
<th>Use case</th>
<th>Why-Not</th>
<th>NedExplainAnswers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crime1</td>
<td>...</td>
<td>m2</td>
</tr>
<tr>
<td>Crime2</td>
<td>m9, m10</td>
<td>P(I.d:692, m9, P.I.d:722, m10)</td>
</tr>
<tr>
<td>Crime3</td>
<td>m9, m10</td>
<td>P(I.d:694, m9, C.I.d:3, m10)</td>
</tr>
<tr>
<td>Crime4</td>
<td>m4, m11</td>
<td>P(I.d:2, m4, G.I.d:146, m11)</td>
</tr>
<tr>
<td>Crime5</td>
<td>m4, m11</td>
<td>P(I.d:2, m4, G.I.d:146, m11)</td>
</tr>
<tr>
<td>Crime6</td>
<td>m4, m5</td>
<td>C.I.d:396, m4, C.I.d:85, m5, C.I.d:112, m4, C.I.d:85, m5</td>
</tr>
<tr>
<td>Crime7</td>
<td>m4</td>
<td>C.I.d:396, m4, C.I.d:85, m4, C.I.d:112, m4, W.I.d:2, m4</td>
</tr>
<tr>
<td>Crime8</td>
<td>m4</td>
<td>P.I.d:2, m4, m5, m6</td>
</tr>
<tr>
<td>Crime9</td>
<td>m4</td>
<td>P.I.d:604, m4, m5</td>
</tr>
<tr>
<td>Crime10</td>
<td>m4</td>
<td>(L.I.d:124, m1), (M.I.d:18, m1)</td>
</tr>
<tr>
<td>Indb1</td>
<td>m4</td>
<td>(L.I.d:124, m1)</td>
</tr>
<tr>
<td>Indb2</td>
<td>m4</td>
<td>(L.I.d:124, m1)</td>
</tr>
<tr>
<td>Gov1</td>
<td>m4</td>
<td>(C.I.d:569, m4, C.I.d:1496, m4)</td>
</tr>
<tr>
<td>Gov2</td>
<td>m4</td>
<td>(C.I.d:1072, m4)</td>
</tr>
<tr>
<td>Gov3</td>
<td>m4</td>
<td>(C.I.d:569, m4)</td>
</tr>
<tr>
<td>Gov4</td>
<td>m4</td>
<td>(S.I.d:8, m4, S.I.d:80, m4, S.I.d:78, m4, S.I.d:79, m4)</td>
</tr>
<tr>
<td>Gov5</td>
<td>m5</td>
<td>(E.I.d:3, m5, E.I.d:212, m5, E.I.d:533, m5)</td>
</tr>
<tr>
<td>Gov6</td>
<td>m4</td>
<td>(null, m4)</td>
</tr>
<tr>
<td>Gov7</td>
<td>m4</td>
<td>(null, m4)</td>
</tr>
<tr>
<td>Gov8</td>
<td>m4</td>
<td>(null, m4)</td>
</tr>
</tbody>
</table>

may hide from the user more specific, but essential information, e.g., in cases where the answer is not a single subquery. For instance, in Crime7, the condensed answer (again not the same as in Why-Not as explained later) identifies m9 and m10 as picky subqueries (refer to Q3 in Fig. 4(b)). From the detailed answer, we moreover obtain the knowledge that there were eleven tuples (originating from Co) for which m9 is picky, but also one tuple (originating from W) for which m9 is picky. This information can be useful, as it indicates that no valid successors of compatible crime tuples reached m9 to join with valid witness tuples. So, the existence of a more detailed answer can be of major help towards understanding the “background” of the result.

NedExplain vs. Why-Not. Our first comparison between NedExplain and Why-Not focuses on the Crime5 use case, with the associated query Q2 (its query tree is given in Fig. 4(a)) having an intermediate empty result on m14 (σ_{sector>9}(C)). The Why-Not algorithm identifies m14 as a picky subquery, which is correct in the sense of responsibility for the missing result tuple. Yet, it cannot be considered as picky in the strict form, since it is not blocking directly any compatible tuple. In this use case, NedExplain provides a more complete and descriptive identify. It identifies m14 as a possible subquery, and in addition it includes m12 in the secondary answer. Knowing that m14 produced an empty set right before the join in m13 can possibly be another reason for m13 being identified as picky.

Let us now focus on the cases where the answers of Why-Not and NedExplain differ, 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 m7 (σ_{C1.type=Assisting}(C1)) as a picky subquery, because it locates the compatible tuples in both C1 and C2. As a result the compatible tuples from C1 with type=Kidnapping are naturally picked at m7. 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, i.e., 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. 4(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 Why-Not algorithm places its compatible source tuples both in P1 and P2. The one coming from P2 does not find any join partners in m12 from P1 as the only candidate ones have names starting with C or D, namely Davenmonet, Chiardola, and Debey. So m12 is picky for the compatible tuple coming from P2. The one coming from P1 survives the selection of m11 and joins with the three persons with equal hair color coming from P2, namely Davenmonet, Chiardola, and Debey. Hence, m12 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 discarded in m11, making m12 a picky subquery for the associated compatible tuple (P2.I.d:51).

Next, we review the Indb2 use case where the associated predicate P is not in its unrefined form. This means that it contains attributes that are not in S_q, but instead were introduced through renamings associated to the subqueries. This use case is based on query Q5 (see Fig. 4(d)), with the following renaming associated to its subquery m2 (join): \( v = (R.name: Christmas Story, L.locationId: USA) \) refers to the new attribute name associated with the subquery m2 \( \sigma_{name} \) in Q5. Thus, before running NedExplain, we transform P to its unrefined form: \( (R.name: Christmas Story, L.locationId: USA) \). 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 m3 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.

Let us now focus on Crime9, based on the SPJA query Q8 (see Fig. 4(e)). As explained in Sec. 3, NedExplain identifies as breakpoint - marked as a bullet on the tree representation - the subquery m2 and the operators of the query are executed in a predefined order (putting selections as near as possible to the breakpoint subquery). In this way, we are able to identify m3 as a why-not answer; the condition of the predicate of this scenario, \( et > 8 \) is satisfied in the input pf m3 \( (am=13) \) but not in its output \( (ct=7) \). Thus, the detailed answer is m3 associated with a null value, since m3 is not picky in a strict sense (valid successors of P.i.d:604 still exist in the output of m3). A variation of this use case is the use case Crime10, where we obtain the detailed answer P.i.d:604, m9, now bound to a specific tuple id. As m9 completely erases the trace of P.i.d:604. Note that Why-Not results are not available (n.a.) as the implementation provided by the authors does not support aggregation and from [2], it is unclear how compatible tuples or the Why-Not question are defined in this case.

Overall, in this first 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 in the context of relational queries. To summarize the highlighted shortcomings, we present the following list:

**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 not to 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.

**Insufficient definition of successors.** 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, and which we model as valid successors. 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.

We defer a more thorough user study to future work.

### 4.3 Runtime Evaluation

Considering runtime, we first study the runtime distribution over the different phases of NedExplain. We then compare NedExplain’s runtime to the runtime of Why-Not.

**Phase-wise runtime.** We distinguish among four phases of NedExplain: (1) Initialization: the global structures initialization; (2) CompatibleFinder: the computation of the compatible tuples set; (3) SuccessorsFinder: the computation of successors of compatible tuples at each subquery output, corresponding to Alg. 3; and (4) Bottom-up traversal: the traversal of the query tree (including SQL query executions) following the bottom-up approach, i.e., Alg. 1 without initialization and call to Alg. 3.

Fig. 5 reports the distribution of the execution time over these phases for each of our use cases. First, we observe a similar distribution for use cases referring to the same query, e.g., Crime1 and Crime2 or GOV1, GOV2, and GOV3, due to the fact that these use cases engage the same database instances and evaluate (almost) the same operators. For use cases involving different queries, we observe that in general, for SPJ queries, the overall runtime is dominated by the initialization phase (between 40 and 77%), essentially caused by the initialization of all relevant Java objects. After initialization, SuccessorsFinder has the second largest impact on overall runtime for most SPJ use cases (GOV1 – GOV3, all based on Q6, and Crime8 being the exceptions). This phase essentially corresponds to Alg. 3, where we compute lineage and do set comparisons. Focusing on the Crime8 exception use case, the last valid successor is lost very early (on m12 in Fig. 4(c)) after evaluating simple operators, which explains both the low fraction of runtime used to find successors but also the low time for the bottom-up traversal. The picture changes when considering SPJ queries, where most of the time is dedicated to the computation of valid successors. This can be explained, by the extra computations needed in the SPJA case of Alg. 3. These computations basically require additional SQL query executions, on the input and output of tree nodes placed after the breakpoint.

**Runtime comparison to Why-Not.** Fig. 6 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, adding a substantial overhead to runtime 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 the future, we plan to more extensively study the impact of...
various parameters on runtime. To conclude, this first set of experiments on runtime indicates that NedExplain can provide high-quality Why-Not answers in reasonable time.

5. CONCLUSIONS AND OUTLOOK

We have addressed the issue of answering why-not questions by first formally defining, for the first time, the domain and concepts of Why-Not questions and Why-Not answers w.r.t. relational queries including projection, selection, join, union, and aggregation. Based on these definitions, we have 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. This set of query based explanations could further be used to obtain modification-based explanations and/or in combination with instance based explanations to calculate hybrid explanations to Why-Not questions.

In the future, besides a more thorough experimental study of algorithm behavior, we plan to extend our algorithm to also consider set difference. This would require 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.

6. REFERENCES


Figure 6: Why-Not and NedExplain execution time