Efficient Computation of Polynomial Explanations of Why-Not Questions
Nicole Bidoit, Melanie Herschel, Aikaterini Tzompanaki

To cite this version:

HAL Id: hal-01182101
https://hal.archives-ouvertes.fr/hal-01182101
Submitted on 6 Aug 2015

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.
Efficient Computation of Polynomial Explanations of Why-Not Questions

Nicole Bidoit
Université Paris Sud / Inria 91405 Orsay Cedex, France nicole.bidoit@lri.fr

Melanie Herschel
Universität Stuttgart
70569 Stuttgart, Germany melanie.herschel
@ipvs.uni-stuttgart.de

Aikaterini Tzompanaki
Université Paris Sud / Inria
91405 Orsay Cedex, France katerina.tzompanaki@lri.fr

ABSTRACT

Answering a Why-Not question consists in explaining why a query result does not contain some expected data, called missing answers. This paper focuses on processing Why-Not questions in a query-based approach that identifies the culprit query components. Our first contribution is a general definition of a Why-Not explanation by means of a polynomial. Intuitively, the polynomial provides all possible explanations to explore in order to recover the missing answers, together with an estimation of the number of recoverable answers. Moreover, this formalism allows us to represent Why-Not explanations in a unified way for extended relational models with probabilistic or bag semantics. We further present an algorithm to efficiently compute the polynomial for a given Why-Not question. An experimental evaluation demonstrates the practicality of the solution both in terms of efficiency and explanation quality, compared to existing algorithms.

Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous

General Terms

Algorithms

Keywords

Why-Not question, explanation, provenance

1. INTRODUCTION

The increasing load of data produced nowadays is coupled with an increasing need for complex data transformations that developers design to process these data in every-day tasks. These transformations, commonly specified declaratively, may result in unexpected outcomes. For instance, given the sample query and data of Fig. 1 on airlines and destination countries, a developer (or traveller) may wonder why Emirates does not appear in the result. Traditionally, she would repeatedly manually analyze the query to identify a possible reason, fix it, and test it to check whether the missing answer is now present or if other problems need to be fixed.

Answering such Why-Not questions, that is, understanding why some data are not part of the result, is valuable in a series of applications, such as query debugging and refinement, data verification or what-if analysis. To help developers explain missing answers, different algorithms have recently been proposed for relational and SQL queries and other types of queries (top-k, reverse skyline).

For relational queries, Why-Not questions can be answered for example based on the data (instance-based explanations), the query (query-based explanations), or both (hybrid explanations). We focus on solutions producing query-based explanations, as these are generally more efficient while providing sufficient information for query analysis and debugging. Essentially, a query-based explanation is a set of query conditions that are responsible for pruning out data relevant to the missing answers. Existing methods producing query-based explanations are not satisfactory as they are designed over query trees, making the explanations depending on the topology of a given tree. Consequently, they return different explanations for the same SQL query and may miss explanations.

Example 1.1. Consider again the SQL query and data of Fig. 1 and assume that a developer wants an explanation for the absence of Emirates from the query result. Fig. 2 shows two possible query trees. It also shows the tree operators that Why-Not [7] (⋆) and NedExplain [3] (⋆) return as query-based explanations as well as the tree operators returned as part of hybrid explanations by Conseil [13, 14] (⋆). Each algorithm returns a different result for each of the two query trees, and in most cases, it is only a partial result as the true explanation of the missing answer is that both the selection is too strict for the tuple (Emirates, 1985, 3) from table Airline and this tuple does not find join partners in table Country.

Figure 1: Example query and data

```
SELECT airline, country
FROM Airline A,
    Country C
WHERE ccode = code
AND year < 1985
```

Figure 2: Reordered query trees for the query of Fig. 1 and algorithm results (Why-Not ⋆, NedExplain ⋆, Conseil ⋆)
The above example clearly shows the shortcomings of existing algorithms. Indeed, the developer first has to understand and reason at the level of query trees instead of reasoning at the level of the declarative SQL query she is familiar with. Second, she always has to wonder whether the explanation is complete, or if there are other explanations that she could consider instead. In this paper, we make the following contributions:

**Extended formalization of Why-Not explanation polynomial.** We have recently introduced polynomials as Why-Not explanations in the context of the relational model under set semantics [2]. A polynomial provides a complete explanation and is independent of a specific query tree representation, solving the problems illustrated by Ex. 1.1. We now extend Why-Not explanations to the relational model under bag and probabilistic semantics. This confirms the robustness of the chosen polynomial representation, making it a good fit for a unified framework for representing Why-Not explanations. In parallel, we considerably simplify our initial framework by eliminating the formerly used notion of query tableaux.

**Efficient Ted++ algorithm.** We show that a naive algorithm computing Why-Not explanations, as presented in [2], is impractical. We thus propose a novel algorithm, Ted++, capable of efficiently computing the Why-Not explanation polynomial, based on techniques like schema and data partitioning (allowing for a distributed computation) and advantageous replacement of expensive database evaluations by mathematical calculations.

**Experimental validation.** We validate our solutions both in terms of efficiency and effectiveness. Our experiments include a comparative evaluation to existing algorithms computing query-based explanations for SQL queries (or sub-languages thereof) as well as a thorough study of Ted++ performance w.r.t. different parameters. Note that such an evaluation was missing from [2].

The remainder of this paper is structured as follows. Sec. 2 covers related work. Sec. 3 defines in detail our problem setting and the Why-Not explanation polynomials. Next, we discuss the Ted++ algorithm in Sec. 4. Finally, we present our experimental setup and evaluation in Sec. 5 and conclude in Sec. 6.

### 2. RELATED WORK

The work presented in this paper falls in the category of data provenance research and specifically on explaining missing answers from query results. Due to the lack of space, we focus here on this sub-problem, thus on algorithms answering Why-Not questions, summarized in Tab. 1. This table classifies the algorithms according to the type of explanation they generate and reports the class of query and Why-Not question (simple or complex) they support.

**Query-based and hybrid explanations.** Why-Not [7] takes as input a simple Why-Not question and a selection, projection, join, and union (SPJU) query and returns the erroneous query operators as query-based explanations. Similarly, NedExplain [3] considers selection, projection, join, and aggregation and unions thereof (SPJUA queries) and simple Why-Not questions as well. The common drawback of the two algorithms is that their design is dependent on a specific query tree representation, thus the explanations proposed are tied to this tree. Moreover, the generated explanations are incomplete. To address this problem, Ted [2] proposes explanations in the form of a polynomial. The shortcomings of this work have already been presented in Sec. 1.

Conseil [13, 14] produces hybrid explanations that include an instance-based and a query-based component. The latter consists in a set of picky query operators. However, as Conseil considers

1A simple Why-Not question involves conditions that impact one relation only, otherwise it is complex (see Sec. 3).

both the data to be possibly incomplete and the query to be possibly faulty, the set of picky query operators associated to a hybrid explanation depends on the set of source edits of the same hybrid explanation.


**Ontology-based explanations.** Cate et al. [6] have recently introduced this type of explanation that is based on external or data-workload generated ontologies. However, they are completely independent of the query to be analyzed.

### 3. WHY-NOT EXPLANATION POLYNOMIAL

This section introduces a polynomial formalization of query-based Why-Not explanations. We assume the reader familiar with the relational model [1], and we only briefly revisit some relevant notions in Sec. 3.1 while we formalize Why-Not questions. In Sec. 3.2, we define the explanation of a Why-Not question as a polynomial. In Sec. 3.3 we provide a unified general framework for Why-Not explanations in the context of set, bag, and probabilistic semantics databases.
Due to lack of space, we do not provide here more real-world examples of Why-Not questions and refer the reader to scenarios in [4]. As we said, a Why-Not question \( WN \) summarizes a set of (missing) tuples that the user expected to find in the query result. To be able to obtain these missing tuples as query results, data from the input relation instances that satisfy \( WN \) need to be combined by the query. The candidate data combinations are what we call compatible tuples and can be computed using \( WN \) as in Def. 3.3.

**Definition 3.3.** (Compatible tuples). **Consider the query** \( Q_{WN} = (S, A(S), WN) \), where \( S \) is the input schema of the query. The set \( CT \) of compatible tuples is the result of the query \( Q_{WN} \) over \( I \).

We further introduce the notion of a well-founded Why-Not question. Intuitively, a Why-Not question can be answered under a query-based approach, only if some data in \( I \) match the Why-Not question (otherwise instance-based explanations should be sought for). Moreover, a Why-Not question is meaningful if it tracks data not already returned by the query.

**Definition 3.4.** (Well-founded Why-Not question). A Why-Not question \( WN \) is said to be well founded if \( CT \neq \emptyset \) and \( \pi_{\tau}(CT) \cap Q[I] = \emptyset \).

**Example 3.2.** Continuing Ex. 3.1, we may wonder why there is not a tuple for which \( R.B < S.D \) and \( T.C \leq 9 \). According to Def. 3.2, this Why-Not question can be seen as the conjunction of the conditions \( R.B < S.D \) and \( T.C \leq 9 \) (Fig. 3(c)). Since \( R.B < S.D \) is a complex condition, \( WN \) is a complex Why-Not question. The compatible tuples set \( CT \) is the result of the query \( Q_{WN} = \pi_{\tau}(CT) \cap Q[I] \), which contains 12 tuples. For example, one compatible tuple is \( \tau = (R.Id:4, R.A:1, R.B:3, S.Id:5, S.D:4, S.E:8, T.Id:8, T.B:3, T.C:4, T.D:5) \).

Each tuple in \( CT \) could have led to a missing tuple, if it was not eliminated by some of the query conditions. Thus, explaining \( WN \) amounts to identifying these blocking query conditions.

### 3.2 Why-Not Explanation

To build the query-based explanation of \( WN \), we start by specifying what explains that a compatible tuple \( \tau \) did not lead to an answer. Intuitively, the explanation consists of the query conditions pruning out \( \tau \).

**Definition 3.5.** (Explanation for \( \tau \)). Let \( \tau \in CT \) be a compatible tuple w.r.t. \( WN \), given \( Q \). Then, the explanation for \( \tau \) is the set of conditions \( E_{\tau} = \{c \in C | c \notin \tau \} \).

**Example 3.3.** Consider the compatible tuple \( \tau_{\text{ex}} \) in Ex. 3.2. The conditions of \( Q \) (see Ex. 3.1), not satisfied by \( \tau_{\text{ex}} \) are: \( c_{1}, c_{3}, \) and \( c_{4} \). So, the explanation for \( \tau_{\text{ex}} \) is \( E_{\tau_{\text{ex}}} = \{c_{1}, c_{3}, c_{4}\} \).

Having defined the explanation w.r.t. one compatible tuple, the explanation for \( WN \) is obtained by simply summing up the explanations for all the compatible tuples in \( CT \), leading to the expression \( \sum_{\tau \in CT} E_{\tau} \) w.r.t. \( WN \). We justify modelling the explanation of \( \tau \) with a product (meaning conjunction) of conditions by the fact that in order for \( \tau \) to ‘survive’ the query conditions and give rise to a missing tuple, every single condition in the explanation must be ‘repaired’. The sum (meaning disjunction) of the products for each \( \tau \in CT \) means that if any explanation is ‘correctly repaired’, the associated \( \tau \) will produce a missing tuple.

Of course, several compatible tuples can share the same explanation. Thus, the final Why-Not explanation is a polynomial having as variables the query conditions and as integer coefficients the number of compatible tuples sharing an explanation.
DEFINITION 3.6 (WHY-NOT EXPLANATION). With the same assumption as before, the Why-Not explanation for \( WN \) is defined as the polynomial

\[
PEX = \sum_{E \in E} \text{coeff} \cdot \prod_{c \in E} c
\]

where \( E = 2^C \) and \( \text{coeff} \in \{0, \ldots, |CT|\} \).

Intuitively, \( E \) contains all potential explanations, and each of these explanations prunes from zero to at most \( |CT| \) compatible tuples. Moreover, an important property of \( PEX \) is the fact that \( \text{coeff} = |\{\tau \in CT|E\} \) is the explanation for \( \tau \), meaning that \( \text{coeff} \) equals the number of compatible tuples with the same explanation.

Each term of the polynomial provides an alternative explanation to be explored by the user who wishes to recover some missing tuples. Additionally, the polynomial offers, through its coefficients, some useful hints to users interested in the number of recoverable tuples. More precisely, by choosing an explanation \( E \) to repair, we obtain an upper bound for the number of compatible tuples that can be recovered. The upper bound is the sum of the coefficients of all the explanations that are sub-sets of the set of conditions of \( E \). Consequently, the coefficients could be used to answer Why-Not questions of the form Why-Not \( S \) missing tuples?

EXAMPLE 3.4. In Ex. 3.3 we found the explanation \( \{c_1, c_3, c_4\} \), which is translated to the polynomial term \( c_1 \cdot c_3 \cdot c_4 \). Taking into consideration all the 12 compatible tuples of our example, we obtain the following PEX polynomial: \( 2 \cdot c_1 \cdot c_1 \cdot c_2 \cdot c_3 \cdot c_1 \cdot c_2 \cdot c_4 \cdot c_3 \cdot c_4 \cdot c_1 + 2 \cdot c_2 \cdot c_1 \cdot c_3 \cdot c_2 \cdot c_3 \cdot c_2 + 2 \cdot c_3 \cdot c_2 \cdot c_3 \cdot c_2 \cdot c_4 \). In the polynomial, each addend, composed by a coefficient and an explanation, captures a way to obtain missing tuples. For instance, the explanation \( c_1 \cdot c_2 \cdot c_4 \) indicates that we may recover some missing answers if \( c_1 \) and \( c_2 \) and \( c_4 \) are changed. Then, the sum of its coefficient 4 and the coefficient 2 of the explanation \( c_1 \cdot c_4 \cdot \{c_1, c_4\} \subseteq \{c_1, c_2, c_4\} \) indicates that we can recover from 0 to 6 tuples.

As the visualization of the polynomial per se may be cumbersome and thus not easy for a user to manipulate, some post-processing steps could be applied. Depending on the application or needs, only a subset of the explanations could be returned like for instance minimum explanations (i.e., for which no sub-explanations exist), or explanations giving the opportunity to recover a specific number of tuples, or have specific condition types etc.

### 3.3 Extension: Bag & Probabilistic Semantics

So far, we have considered databases under set semantics only.

In this section, we discuss how the definition of Why-Not explanation (Def. 3.6) extends to settings with conjunctive queries over bag and probabilistic semantics.

\( K \)-relations, as introduced in [10], capture in a unified manner relations under set, bag, and probabilistic semantics. Briefly, tuples in a \( K \)-relation are annotated with elements in \( K \). In our case, we consider that \( K \) is a set of unique tuple identifiers, similar to our special attribute \( R.Id \) in Sec. 3.1.

In what follows, we use the notion of how-provenance of tuples in the result of a query \( Q \). The how-provenance of \( t \in \text{Range}(Q) \) is modelled as the polynomial obtained by the positive algebra on \( K \)-relations, proposed in [10]. Briefly, each \( t \) is annotated with a polynomial where variables are tuple identifiers and coefficients are natural numbers. Roughly, if \( t \) results from a selection operator on \( t_1 \) annotated with \( Id_1 \), then \( t \) is also annotated with \( Id_1 \). If \( t \) is the result of the join of \( t_1 \) and \( t_2 \), then \( t \) is annotated with \( Id_1.Id_2 \).

We compute the generalized Why-Not explanation polynomial \( PEX_{gen} \) as follows. Firstly, we compute the how-provenance for compatible tuples in \( CT \) by evaluation of the query \( Q_{WN} \) (Def. 3.3) w.r.t. the algebra in [10]. Recall that \( Q_{WN} \) contains only selection and join operators. Thus, each compatible tuple \( \tau \in CT \) is annotated with its how-provenance polynomial, denoted by \( \eta_\tau \).

Then, we associate the expressions of how and why-not provenance. In order to do this, for each compatible tuple \( \tau \in CT \), we combine its how-provenance polynomial \( \eta_\tau \) with its explanation \( E_\tau \) (Def. 3.5). So, each \( \tau \) is associated with the expression \( \eta_\tau.E_\tau \).

Finally, we sum the combined expressions for all compatible tuples, which leads to the expression \( \sum_{\tau \in CT} \eta_\tau.E_\tau \).

We now briefly comment on how \( PEX_{gen} \) is instantiated to deal either with the set, bag, or probabilistic semantics. Indeed, the ‘specialization’ of \( PEX_{gen} \) relies on the interpretation of the elements in \( K \), that is on a function \( Eval \) from \( K \) to some set \( L \). For the set semantics, \( E \) is chosen as the set of natural numbers \( \mathbb{N} \) and each tuple identifier is mapped to its number of occurrences. Finally, for probabilistic databases, \( L \) is chosen as the interval \([0, 1]\) and each tuple identifier is mapped to its occurrence probability.

Thus, the generalized definition of Why-Not explanation is parametrized by the mapping \( Eval \) of the annotations (elements in \( K \)) in the set \( L \).

DEFINITION 3.7. (Generalized Why-Not explanation polynomial)

Given a query \( Q \) over a database schema \( S \) of \( K \)-relations, the generalized Why-Not explanation polynomial for \( WN \) is

\[
PEX_{gen} = \sum_{E \in E} \left( \sum_{\tau \in CT \times L, \eta_\tau.E_\tau = E} \text{Eval}(\eta_\tau) \right) E
\]

where \( E = 2^C \), \( \eta_\tau \) is the how-provenance of \( \tau \), and \( Eval:K \rightarrow L \) maps the elements of \( K \) to values in \( L \).

### 4. TED++ ALGORITHM

The naive Ted algorithm [2] implements the definitions of [2] for Why-Not explanations in a straightforward manner. Essentially, Ted first enumerates the set of compatible tuples. Then, it computes the explanation for each compatible tuple, leading to the computation of the final Why-Not explanation. However, both of these steps make Ted computationally prohibitive. Not only is the computation of the set of compatible tuples time and space consuming as it often requires cross product executions, but the same holds for the iteration over this (potentially very large) set. Ted’s time complexity is \( O(n^{3k}) \), where \( n = \max(|\{Id_1\}|, |R|) \), \( R \in S \). As experiments in Sec. 5 confirm, this complexity renders Ted impractical.

To overcome Ted’s poor performance, we propose Ted++. The main feature of Ted++ is to completely avoid enumerating and iterating over the set \( CT \); thus it significantly reduces both space and time consumption. Instead, Ted++ opts for (i) iterating over the space of possible explanations, which is expected to be much smaller, (ii) computing partial sets of passing compatible tuples, and (iii) computing the number of eliminated compatible tuples for each explanation. Intuitively, passing tuples w.r.t. an explanation are tuples satisfying the conditions of the explanation. Finally, we compute the polynomial based on mathematical calculations.

Alg. 1 provides an outline of Ted++. The input includes the query \( Q = (S, \Gamma, C) \), the Why-Not question \( WN \) and the input instance \( I \). Firstly in Alg. 1, line 1, all potential explanations (combinations of the conditions in \( C \)) are enumerated (\( E = 2^C \)). The remaining steps, discussed in the next subsections, aim at computing the coefficient of each explanation. To illustrate the concepts
Algorithm 1: Ted++

Input: $Q=(S, \Gamma, C), I, WN$

Output: PEX

1. $E \leftarrow \text{powerset}(C)$;
2. $P \leftarrow \text{validPartitionings}(S, WN)$; * Def. 4.1 *
3. for Part in $P$ do
4. $CT_{\text{Part}} \leftarrow (\text{Part}, A(\text{Part}), WN_{\text{Part}})[I_{\text{Part}}]$;
5. $\text{coefficientEstimation}(E, \text{Partition})$;
6. $\text{PEX} \leftarrow \text{post-processing}(E, \text{Partion})$; * Eq. (F) *
7. return PEX.

4.1 Partial Compatible Tuples Computation

Using the conditions in $WN$, Ted++ partitions the schema $S$ (Alg. 1, line 2) into components of relations connected by the conditions in $WN$ (Def. 4.1).

**Definition 4.1.** (Valid Partitioning of $S$). Given $WN$, the partitioning of a database schema $S$ into $k$ partitions, denoted $\mathcal{P} = \{\text{Part}_1, \ldots, \text{Part}_k\}$, is valid if each $\text{Part}_i$, $i \in \{1, \ldots, k\}$ is minimal w.r.t. the following property:

- if $R \in \text{Part}_i$, and $R' \in S$ s.t. $\exists c \in WN$ with $A(c) \cap A(R') = \emptyset$ and $A(c) \cap A(R) \neq \emptyset$ then $R' \in \text{Part}_i$.

The partitioning of $S$ allows for handling compatible tuples more efficiently, by "cutting" them in distinct meaningful ‘chunks’. We refer to chunks of compatible tuples as **partial** compatible tuples and group them in sets depending on the partition they belong to.

The set $CT_{\text{Part}}$, where $\text{Part} \in \mathcal{P}$ is evaluated by evaluating the query $Q_{\text{Part}} = (\text{Part}, A(\text{Part}), WN_{\text{Part}})$ over $I_{\text{Part}}$ (Alg. 1, line 4). $WN_{\text{Part}}$ and $I_{\text{Part}}$ denote the restriction of $WN$ and $I$ over the relations in $\text{Part}$, respectively.

**Example 4.1.** The valid partitioning of $S$ is $\text{Part}_1 = \{R, S\}$ (because of the condition $R.B \approx S.D$) and $\text{Part}_2 = \{T\}$. The sets of partial compatible tuples $CT_{\text{Part}_1}$ and $CT_{\text{Part}_2}$ are given in the bottom line of Fig. 4.

It is easy to prove that the valid partitioning of $S$ is unique and that the set $CT$ can be computed from the sets $CT_{\text{Part}}$.

**Lemma 4.1.** Let $\mathcal{P}$ be the valid partitioning of $S$. Then, $CT = \times_{\text{Part} \in \mathcal{P}} [CT_{\text{Part}}]$.

Lemma 4.1 makes it clear how to compute $CT$ from partial compatible tuples. Our algorithm is designed in a way that avoids computing $CT$ and relies on the computation of $CT_{\text{Part}}$ only.

Next, we compute the number of compatible tuples pruned by each potential explanation, using the partial compatible tuple sets. In this way we calculate the coefficient of the terms in the polynomial. Since from this point on we are only handling compatible tuples, we omit the word ‘compatible’ to lighten the discussion.

4.2 Polynomial Coefficient Estimation

Each set $E$ in the powerset $E$ is in fact a potential explanation that is further processed. This process is meant to associate with $E$ (i) the set of partitions $\text{part}_E$ on which $E$ is defined, (ii) the view definition $V_E$ meant to store the passing partial tuples w.r.t. $E^2$, and (iii) the number $\alpha_E$ of tuples eliminated by $E$.

Alg. 2 describes how we process $E$ in ascending order of explanation size. This enables us to reuse results obtained for sub-explanations and in combination with mathematics, avoid cross product computations.

We first determine the set of partitions for an explanation $E$ as $\text{part}_E = \cup_{\alpha \in E} \{\text{Part}_1\}$, where $\text{Part}_1$ contains at least one relation over which $c$ is specified.

**Example 4.2.** Consider $E_1 = \{c_1\}$ and $E_2 = \{c_2\}$. From Fig. 3(b) and the partitions in Fig. 4, we can see that $c_1$ impacts only $\text{Part}_1$, whereas $c_2$ spans over $\text{Part}_1$ and $\text{Part}_2$. Hence, $\text{part}_{E_1} = \{\text{Part}_1\}$ and $\text{part}_{E_2} = \{\text{Part}_1, \text{Part}_2\}$. Then, $\mathcal{E} = \{c_1, c_2\}$ is impacted by the union of $\text{part}_{E_1}$ and $\text{part}_{E_2}$, thus $\text{part}_{E} = \{\text{Part}_1, \text{Part}_2\}$.

We use Eq. (A) to calculate the number $\alpha_E$ of eliminated tuples, using the number $\beta_E$ of eliminated partial tuples and the cardinality of the partitions not in $\text{part}_E$.

**Intuitively, this formula extends the partial tuples to "full" tuples over $CT$'s schema.**

$$\alpha_E = \beta_E \times \prod_{\text{Part} \in \mathcal{P} \setminus \text{part}_E} |CT_{\text{Part}}|,$$  \hspace{1cm} (A)

where $\text{part}_E = \mathcal{P} \setminus \text{part}_E$. Note that when $\text{part}_E$ is empty, we abusively consider that $\prod = 1$.

The presentation now focuses on calculating $\beta_E$. Two cases arise depending on the size of $E$.

**Atomic explanations.** We start with atomic explanations $E$ containing only one condition $c$ (Algorithm 2 lines 3-5). We first compute the set of passing partial tuples w.r.t. $c$, i.e., the tuples that satisfy $c$, which we store in the view $V_c$:

$$V_c = \begin{cases} \pi_{(R_{id} \mid R \in \text{Part}_1)}(\sigma_{c}(CT_{\text{Part}_1})) & \text{if } \text{part}_E = \{\text{Part}_1\} \\ \pi_{(R_{id} \mid R \in \text{Part}_1 \cup \text{Part}_2)}(\sigma_{c}(CT_{\text{Part}_1}) \cap \sigma_{c}(CT_{\text{Part}_2})) & \text{if } \text{part}_E = \{\text{Part}_1, \text{Part}_2\} \end{cases}$$

We choose to store passing rather than eliminated tuples as they are usually less numerous. In an optimized version this decision could be made dynamically based on view cardinality estimation.
Then, the number of eliminated partial tuples by $E$ is

$$
\beta_E = \prod_{p \in Part_E} |CT_{|p_{\text{part}}}| - |V_c| \tag{B}
$$

**Example 4.3.** For $c_3$, we have $p_{\text{part}} = \{ Part_1, Part_2 \}$, so $\{V_{23} = R_{14, 5, 6, 7} \cdot \text{Id}(V_{24}) \} \cdot |CT_{|p_{\text{part}}}|$. This results in $\{V_{23} = 4\}$, and by Eq. (B) we obtain $\beta_{c_3} = |CT_{|p_{\text{part}}}| - |V_c| = 3 \cdot 4 - 4 = 8$. Since all partitions of $\mathcal{P}$ are in part, applying Eq. (A) results in $\alpha_{c_3} = 2\beta_{c_3} = 8$. For $c_5$, $\beta_{c_5} = |CT_{|p_{\text{part}}}| - |V_c| = 4 - 2 = 2$, so $\alpha_{c_5} = 3 \cdot 2 = 6$. Fig. 4 (second level) displays the process for all atomic explanations.

**Non atomic explanations.** Now, assume that $E = \{ c_1, \ldots, c_n \}$, $n > 1$ (Alg. 2, lines 6-16). For the moment, we assume that the conditions in $E$ share the same schema, so the intersection and union of $V_{2i}$ for $i = 1, \ldots, n$ are well-defined. Firstly, we compute the view $V_{2i}$ of passing partial tuples w.r.t. $E$ as $V_{2i} = V_{c_i} \cap \ldots \cap V_{c_n}$. To compute the number of partial tuples pruned out by $E$, we need to find the number of partial tuples pruned out by $c_1$ and ... and $c_n$, i.e., $\beta_E = \sum_{\theta \in J[n]} (-1)^{|J|+1} \sum_{j \in J} |V_{2j}|$.

By the well-known Demorgan law [22], we have $\beta_E = V_{23} \cup \ldots \cup V_{2n}$, which spares us from computing the complements of $V_{2i}$.

To compute the cardinality of the union of the $V_{2i}$, we rely on the Principle of Inclusion and Exclusion for counting [11]:

$$
\sum_{i=1}^{n} |V_{2i}| = \sum_{\theta \in J[n]} (-1)^{|J|+1} \sum_{j \in J} |V_{2j}|
$$

We further rewrite the previous formula to result in a refined formula for sub-combinations of $E$, obtaining Eq. (C).

$$
\sum_{i=1}^{n} |V_{2i}| = \sum_{i=1}^{n-1} |V_{ci}| + \sum_{\theta \in J[n-1]} (-1)^{|J|} \sum_{j \in J} |V_{cj} \cap V_{cn}|
$$

At this point, we can compute $\beta_E$. However, so far we assumed that the conditions in $E$ share the same condition. In the general case, this does not hold and we have to "extend" the schema of a view $V_{2i}$ to the one of $V_c$, in order to ensure set operations to be well-defined. The cardinality of an extended $V_{2i}^{\text{ext}}$ is given by Eq. (D).

$$
|V_{2i}^{\text{ext}}| = \prod_{p \in \text{Part}\setminus c} |CT_{\text{Part}}| \cdot |V_{c}|
$$

Based on Eq. (D) we obtain Eq. (E) that generalizes Eq. (C).

$$
\left( \bigcup_{i=1}^{n} V_{2i} \right)^{\text{ext}} = \left( \bigcup_{i=1}^{n-1} V_{ci} \right)^{\text{ext}} + |V_{c}^{\text{ext}}| + \sum_{\theta \in J[n-1]} (-1)^{|J|} \left( \left( \bigcup_{i \in J} V_{ci} \right) \cap V_{cn} \right)^{\text{ext}}
$$

In Eq. (E) we have replaced the intersection with natural join. The cardinalities of the views $V^{\text{ext}} = \bigcup_{i=1}^{n} V_{2i}^{\text{ext}}$ associated with $E'$ for $|J| < n - 1$, have already been computed by previous steps and have only to be extended to the schema of $V_{2i}$. When $|J| = n - 1$, then $V_{2i}^{\text{ext}} = V_{2i}$. A detailed discussion on how and when we materialize the view $V_{2i}$ is given shortly after.

Now, we trivially compute the number $\beta_E$ of eliminated partial tuples as the complement of $\left( \bigcup_{i=1}^{n} V_{2i} \right)^{\text{ext}}$ (Alg. 2, line 17). The number of ‘full’ eliminated tuples is then calculated by Eq. (A).

**Example 4.4.** To illustrate the concepts introduced above, please follow on Fig. 4 below discussion.

For the explanation $c_2c_3$, Eq. (E) gives: $\left( V_2 \cup V_3 \right)^{\text{ext}} = |V_2^{\text{ext}}| + |V_3^{\text{ext}}| - |V_2 \cap V_3^{\text{ext}}|$. The schema of $c_2c_3 = \{ Part_1, Part_2 \}$ is $\Gamma_{c_2c_3} = \{ R_{14, 5, 6, 7} \cdot \text{Id}(V_{24}) \}$. The view $V_{23}$ has already a matching schema, thus $V_{23}^{\text{ext}} = |V_{23}| = 4$. For $V_{24}$, $\Gamma_{c_3} = \{ T_{14, 7} \}$, we thus apply Eq. (D) and obtain $|V_{24}^{\text{ext}}| = |V_{24}^{\text{ext}}| |V_{24}^{\text{ext}}| = 3 \cdot 2 = 6$. Still, $\beta_{23} = |V_2 \cap V_3^{\text{ext}}|$ remains to be computed. Intuitively, because $V_2$ and $V_3$ target schemas share attribute $T_{14}$, $V_{23} = V_2 \cap V_3 \cap T_{14} V_4$. The view $V_4$ is materialized and contains 2 tuples (as shown in Fig. 4). So, finally, from Eq. (E) we obtain $|V_2 \cup V_3| = 6 - 0 = 6$. Since $|\{ Part_1 \}| |\{ Part_2 \}| = 12$ then $\beta_{23} = 12 - 8 = 4$, and by Eq. (A) $\alpha_{c_3} = 4$.

We now focus on the explanation $c_2c_5$. The schemas of $V_3$ and $V_5$ are disjoint and intuitively $V_{35} = V_3 \cap V_5$. Here, $V_{35}$ is not materialized, we simply calculate $|V_3| = |V_3| = 0$. Then, $\beta_{35} = 12 - 10 = 2$. As we will see later, these steps are never performed in our algorithm. The fact that $c_5$ does not eliminate any tuple (see $\alpha_{c_5} = 0$ in Fig. 4) implies that neither do any of its super-combinations. Thus, a priori we know that $\alpha_{c_5} = 0$.

Finally, we illustrate the case of a bigger size combination, for example $c_2c_3c_4$ of size 3. Eq. (E) yields $\left( V_2 \cup V_3 \cup V_4 \right)^{\text{ext}} = |V_2 \cup V_3|^{\text{ext}} + |V_4^{\text{ext}}| - |V_2 \cap V_3^{\text{ext}}| - |V_4^{\text{ext}}| - |V_2 \cap V_3^{\text{ext}}| + |V_2 \cap V_3 \cap V_4^{\text{ext}}|$. All terms of the right side of the equation are available from previous iterations, except for $|V_2 \cap V_3 \cap V_4^{\text{ext}}|$. As before, we check the common attributes of the views and obtain $\{ V_2 \cup V_3 \cup V_4 \} = 6 - 0 = 6$. When this partitioning is a singleton, $V_{24}$ needs to be materialized (Alg. 2, line 9). The materialization of $V_{24}$ is specified by joining the views associated with the sub-conditions, which may be done in more than one way, as usual. For example, for the combination $c_2c_3c_4$, $V_{24}$ can be either computed through $V_2 \cap V_3 \cap V_4$ or $V_2 \cap V_3 \cap V_4$ or $V_2 \cap V_3 \cap V_4$... because all these views are known from previous iterations. The choice of the query used to materialize $V_{24}$ is done based on a cost function. This function gives priority to materializing $V_{24}$ by means of one join, which is always possible: because $V_{24}$ needs to be materialized, we know that at least one view associated with a sub-combination of size $n - 1$ has been materialized. In other words, priority is given to using at least one materialized view associated with one of the largest sub-combinations. For our example, it means that either $V_2 \cap V_3 \cap V_4$ or $V_2 \cap V_3 \cap V_4$ or $V_2 \cap V_3 \cap V_4$... is considered. In order to choose among the one-join queries computing $V_{24}$, we favour a one-join query $V_2 \cap V_3 \cap V_4$ minimal w.r.t. $|V_2| + |V_3|$. For the example, and considering also Fig. 4 we find that $|V_2| + |V_3| = |V_4| + |V_{23}| = 5$ and $|V_2| + |V_{23}| = 3$. So, the query used for the materialization is $V_2 \cap V_3 \cap V_4$ (its result being empty in our example). Nevertheless, we avoid the materialization of $V_{24}$ if the partitioning is a singleton (Alg. 2, line 9).
& 16), when for some sub-combination $E'$ of $E$ it was computed that $\alpha_{E'}=0$. In that case, we know a priori that $\alpha_E=0$ (see Ex. 4.4).

The partitioning is not a singleton, $V_E$ is not materialized (Alg. 2, line 14). For example, the partitioning for $c_3, c_5$ is not a singleton and so the size $|V_E|=|V_5|\times|V_5|=6$.

**Post-processing.** In Alg. 2 we associated with each possible explanation $E$ the number of eliminated tuples $\alpha_E$. However, $\alpha_E$s includes any tuple eliminated by $E$, even though the same tuples may be eliminated by some super-combinations of $E$ (see Ex. 4.5). This means that for some tuples, multiple explanations have been assigned. To make things even, the last step of Ted++ (Alg. 1, line 6) is about calculating the coefficient of $E$ by subtracting the coefficients of its super-combinations from $\alpha_E$:

$$\text{coef}_{E} = \alpha_E - \left( \sum_{E' \subseteq E} \text{coef}_{E'} \right) \quad (F)$$

**Example 4.5.** Consider known $\text{coef}_{f_{1234}}=2$ and $\text{coef}_{f_{123}}=2$. We have found in Ex. 4.4 that $\alpha_{E_2}=4$. With Eq. (F), $\text{coef}_{E_2}=4-2-2=0$. In the same way $\text{coef}_{E_3}=4-0-2-2=0$. The algorithm leads to the expected Why-Not explanation polynomial already provided in Ex. 3.4.

### 4.3 Complexity Analysis

In the pseudo-code provided by Alg. 1, we can see that Ted++ divides into the phases of (i) partitioning $S$, (ii) materializing a view $V_E$ for each partition, (iii) computing the explanations, and (iv) computing the exact coefficients. When computing the explanations, according to Alg. 2, Ted++ iterates through $2^{[C]}$ condition combinations and for each, it decides upon view materialization (again through partitioning) before materializing it, or simply calculates $|V_E|$ before applying equations to compute $\alpha_E$. Overall, we consider that all mathematical computations are negligible so, the worst case complexities of steps (i) through (iv) sum up to $O(|S|+|W|)$ + $O(|S|+|C|)$ + $O(2^{|C|})$. For sufficiently large queries, we can assume that $|S|+|C|<<2^{|C|}$, in which case the complexity simplifies to $O(2^{|C|})$.

The complexity analysis above does not take into account the cost of actually materializing views; in its simplified form, it only considers how many views need to be materialized in the worst case. Assume that $n=max\{|I_R|\mid R \in S\}$. The materialization of any view is bound by the cost of materializing a cross product over the relations involved in the view - in the worst case $O(n^{2k})$. This yields a combined complexity of $O(2^{|C|}n^{2k})$. However, Ted++ in the general case (more than one induced partitions), has a tighter upper bound: $O(n^{k_E+1} + n^{k_E2} + \cdots + n^{k_EN})$, where $k_E=|\text{part}_E|$, for all combinations $E$ and $N=2^{|C|}$.

### 5. EXPERIMENTAL EVALUATION

We perform an experimental evaluation of Ted++ on real and synthetic datasets. In Sec. 5.1, we compare Ted++ to Ted [2], NedExplain [3], and Why-Not [7]. Sec. 5.2 studies the runtime of Ted++ w.r.t. various parameters that we vary in a controlled manner. All Java implementations of the algorithms ran on MAC OS X 10.9.5 with 1.8 GHz Intel Core i5, 4GB memory, and 120GB SSD. PostgreSQL 9.3 was used as database system.
5.1 Comparative Evaluation

The comprehensive comparative evaluation of Why-Not and NedExplain considers both efficiency (runtime) and effectiveness (explanation quality). When considering efficiency, we also include Ted in the comparison (Ted producing the same Why-Not explanation as Ted++).


For each dataset, we have created a series of scenarios (crime1-gov5 in Tab. 3 - ignore remaining scenarios for now). Each scenario consists of a query further defined in Tab. 2 (Q1-Q7) and a simple Why-Not question, as Why-Not and NedExplain support only this type of Why-Not question. We have designed queries with a small set of conditions (Q6) or a larger one (Q1, Q3, Q5, Q7), containing inequalities (Q2, Q4, Q5, Q6).

5.1.1 Why-Not Explanation Evaluation

Tab. 1 states that the explanations returned by Why-Not and NedExplain consist of sets of query conditions, whereas Ted++ returns a single set of conditions. For comparison purposes, we trivially map Ted++’s Why-Not explanation to sets of conditions, e.g., \(3c_3 \land c_4 \land c_6 \land c_8\) maps to \(\{c_1, c_4\}, \{c_3\}\). For conciseness, we abbreviate condition sets, e.g., \(c_{34, c_{16}}\).

Tab. 4 summarizes the Why-Not explanations of the three algorithms. These scenarios make apparent that the explanations by NedExplain or Why-Not are incomplete, in two senses. First, they produce only a subset of the possible explanations, failing to provide alternatives that could be useful to the user when she tries to fix the query. Second, even the explanation they provide may lack parts, which can drive the user to fruitless fixing attempts. On the contrary, Ted++ produces all the possible, complete explanations.

For the first argument, consider the scenario gov2. Why-Not and NedExplain return \(c_1\) and \(c_3\) respectively, but they both fail to indicate that both the explanations are valid, as opposed to Ted++.

For the second argument, consider crime5. NedExplain returns \(c_1\) and \(c_4\), the explanation of Ted++ does not contain the atomic explanation \(c_1\), but there exist combinations including \(c_1\) as a part, like \(c_{15}\). This means that the explanation by NedExplain is incomplete; a repair attempt of \(c_1\) alone will never yield the desired results. Similarly, crime7 illustrates a case, where the Why-Not algorithm produces an explanation \(c_1\) that misses some parts.

For the third argument, consider crime8. NedExplain returns \(c_1\) and \(c_4\), the explanation of Ted++ does not contain the atomic explanation \(c_1\), but there exist combinations including \(c_1\) as a part, like \(c_{15}\). This means that the explanation by NedExplain is incomplete; a repair attempt of \(c_1\) alone will never yield the desired results. Similarly, crime7 illustrates a case, where the Why-Not algorithm produces an explanation \(c_1\) that misses some parts.

5.1.2 Runtime Evaluation

Ted++ vs. NedExplain and Why-Not. Fig. 5 shows the runtimes in logarithmic scale for each algorithm and scenario. We observe that Ted++ and NedExplain are comparable and that in some cases, Ted++ is significantly faster than Why-Not.

Ted++ traces compatible tuples based on tuple lineage stored in Trio. As already stated in [3, 7], this design choice slows down Why-Not. On the contrary, both NedExplain and Ted++ compute
Table 4: Ted++, Why-Not, NedExplain answers per scenario

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Ted++</th>
<th>Why-Not</th>
<th>NedExplain</th>
</tr>
</thead>
<tbody>
<tr>
<td>crime1</td>
<td>c1</td>
<td>c1</td>
<td>c1</td>
</tr>
<tr>
<td>crime2</td>
<td>c2</td>
<td>c2</td>
<td>c2</td>
</tr>
<tr>
<td>crime3</td>
<td>c3</td>
<td>c3</td>
<td>c3</td>
</tr>
<tr>
<td>crime4</td>
<td>c4</td>
<td>c4</td>
<td>c4</td>
</tr>
<tr>
<td>crime5</td>
<td>c5</td>
<td>c5</td>
<td>c5</td>
</tr>
<tr>
<td>crime6</td>
<td>c6</td>
<td>c6</td>
<td>c6</td>
</tr>
<tr>
<td>crime7</td>
<td>c7</td>
<td>c7</td>
<td>c7</td>
</tr>
<tr>
<td>crime8</td>
<td>c8</td>
<td>c8</td>
<td>c8</td>
</tr>
<tr>
<td>imdb1</td>
<td>c9</td>
<td>c9</td>
<td>c9</td>
</tr>
<tr>
<td>imdb2</td>
<td>c10</td>
<td>c10</td>
<td>c10</td>
</tr>
<tr>
<td>gov1</td>
<td>c11</td>
<td>c11</td>
<td>c11</td>
</tr>
<tr>
<td>gov2</td>
<td>c12</td>
<td>c12</td>
<td>c12</td>
</tr>
<tr>
<td>gov3</td>
<td>c13</td>
<td>c13</td>
<td>c13</td>
</tr>
<tr>
<td>gov4</td>
<td>c14</td>
<td>c14</td>
<td>c14</td>
</tr>
<tr>
<td>gov5</td>
<td>c15</td>
<td>c15</td>
<td>c15</td>
</tr>
</tbody>
</table>

Figure 5: Runtime for Ted++, Ted, NedExplain, and Why-Not compatible data more efficiently. We claim that a better implementation choice for tuple tracing in Why-Not would yield a runtime comparable to NedExplain, a claim backed up by their comparable runtime complexities. Another problem of NedExplain and Why-Not lies in the choice to trace compatible data w.r.t. tuples from the input relations but not necessarily compatible ones.

Let us see what happens when Ted++ is slower than - but still comparable to - NedExplain, for example in govt1-govt3. In NedExplain all compatible tuples are pruned out by conditions very close to the leaf level of the query tree, so the bottom-up traversal of the tree can stop very early. Ted++ always “checks” all conditions so cannot benefit from such an early termination. However, this runtime improvement of NedExplain often comes at the price of incomplete explanations (e.g., govt1).

**Ted++ vs. Ted.** Fig. 5 reports runtimes for Ted on 6 out of 15 scenarios as for the others, Ted runs out of time. To examine this behavior, we compare the time distribution in Ted and Ted++(Fig. 6). The algorithms are divided in four common phases. Note that in scenarios crime7, govt1 – govt3 the diagram bars for Ted are not totally displayed as the execution time is much higher compared to the other scenarios and to the runtime of Ted++ (the runtime of the coefficient Estimation phase is the label on the respective bars).

As said in Sec. 4, Ted’s main issue is its dependence on the number of compatible tuples. This is experimentally observed in Fig. 6: with the growth of the set of compatible tuples, the time dedicated to coefficient Estimation also grows (the scenarios are reported in an ascending order of number of compatible tuples). The number of compatible tuples affects Ted++ too, but not as much. This can be seen in crime8 and crime7, or govt3 and govt1; while the number of tuples grows, Ted++’s runtime remains roughly steady.

**5.2 Ted++ Analysis**

We now study Ted++’s behavior w.r.t. the following parameters: (i) the type (simple or complex) of the input query $Q$ and the number of $Q$’s conditions, (ii) the type of the Why-Not question (simple or complex) and the number and selectivity of conditions the Why-Not question involves, and (iii) the size of the database instance $T$. Note that (ii) and (iii) are tightly connected with the number of compatible tuples, which is one of the main parameters influencing the performance. Another important factor is the selectivity of the query conditions over the compatible data.

For the parameter variations (i) and (ii), we use again the crime, imdb, and gob databases. To adjust the database instance size for case (iii), we use data produced by the TPC-H benchmark data generator (http://www.tpc.org/tpch/). We have generated instances of 1GB and 10GB and further produced smaller data sets of 10MB and 100MB to obtain a series of datasets whose size differs by a factor of 10. In this paper, we report results for the original query Q3 of the TPC-H set of queries. It includes two complexes and three simple conditions, two of which are inequality conditions. Since the original TPC-H query Q3 is an aggregation query, we have changed the projection condition. The queries used in this section are Q1, Q3, Q1pch (Tab. 2) and the scenarios are crime1-tpch (Tab. 3).

**Adjusting the query.** Given a fixed database instance and Why-Not question, we start from query Q1 and gradually add simple conditions, yielding the series of queries Q1, Q2, Q3, Q4. The evolution of Ted++ runtime for these queries is shown in Fig. 7 (a). Similarly, starting from query Q1, we introduce step by step complex conditions, yielding Q1, Q2, Q3. Corresponding runtime results are reported in Fig. 7 (b).

As expected, in both cases, increasing the number of query conditions (either complex or simple) results in increasing runtime. The incline of the curve depends on the selectivity of the introduced condition; the less selective the condition the steeper the line becomes. This is easy to explain, as the view for the explanations involving a low selective condition contains more tuples (spassing partial tuples). This, leaves space for further optimization by dynamically deciding on passing vs eliminated tuples materialization.

**Adjusting the Why-Not question.** The scenarios considered for Fig. 8 (a) have as starting point the simple Why-Not question of crime5 (see Tab. 3). Keeping the same input instance and query, we add attribute-constant comparisons (i.e., simple conditions) to WN, resulting in fewer tuples in each step. As expected, the more conditions (the less tuples) the faster the Why-Not explanation is returned, until we reach a certain point (here from crime5c3 on).
From this point, the runtime is dominated by the time to communicate with the database that is constant over all scenarios.

In Fig. 8 (b) we examine complex Why-Not questions. As we add complex conditions in a Why-Not question, the number of generated partitions (potentially) drops as more relations are included in a same partition. To study the impact of the induced number of partitions in isolation, we keep the number of the compatible tuples constant in our series of complex scenarios (\(imdb_{c2}, imdb_{c3}\)). The number of partitions entailed by \(imdb_{c2}, imdb_{c3}\), and \(imdb_{c3}\) are 3, 2, and 1, respectively. The results of Fig. 8 (b) confirm our theoretical complexity discussion, i.e., as the number of partitions decreases, the time needed to produce the Why-Not explanation increases.

Increasing size of input instance. Now we increase the database size for scenarios with one simple or one complex Why-Not question \(WN\), over the same query \(Q_{topk}\). The simple \(WN\) includes two inequality conditions, in order to be able to compute a reasonable number of compatible tuples. The complex \(WN\) contains one complex condition, one inequality simple condition and one equality simple condition. It thus represents an average complex Why-Not question, creating two partitions over three relations.

Fig. 9 (a) shows the runtimes for both scenarios. The increasing runtime is tightly coupled to the fact that the number of computed tuples is augmenting proportionally to the database size, as shown in Fig. 9 (b). We observe that for small datasets (<500MB) in the complex scenario, \(Ted++\)’s performance decreases with a low rate, whereas the rate is higher for larger datasets. For the simple scenario, runtime deteriorates in a steady pace. This behavior is aligned with the theoretical study, when the number of partitions is decreasing the complexity rises.

In summary, our experiments have shown that \(Ted++\) generates a more useful and complete Why-Not explanation than the state of the art. Moreover, \(Ted++\) is competitive in terms of runtime. The dedicated experimental evaluation on \(Ted++\) verifies that it can be used in a large variety of scenarios with different parameters. Finally, the fact that the experiments were conducted on an ordinary laptop supports \(Ted++\)’s feasibility.

6. CONCLUSION AND OUTLOOK

This paper provides a framework for Why-Not explanations based on polynomials, which enables to consider relational databases under set, bag and probabilistic semantics in a unified way. To efficiently compute the Why-Not explanation polynomial under set semantics we have designed a new algorithm \(Ted++\), whose main feature is to completely avoid enumerating and iterating over the set of compatible tuples, thus significantly reducing both space and time consumption. Our experimental evaluation showed that \(Ted++\) is at least as efficient as existing algorithms while providing useful insights in its Why-Not explanation for a developer. Also, we show that \(Ted++\) scales well w.r.t various parameters, making it a practical solution.

Why-Not explanation polynomials are easy to extend for unions of conjunctive queries, whereas an extension for aggregation queries is subject to future work. Currently, we have been working on exploiting the Why-Not explanation polynomial to efficiently rewrite a query in order to include the missing answers in its result set. As there are many rewriting possibilities, we plan to select the most promising ones based on a cost function, built with the polynomial. For instance, we may rank higher rewrites with minimum condition changes (i.e., small combinations), minimum side-effects (i.e., small coefficients), etc.

7. REFERENCES