Efficient computation of polynomial explanations of Why-Not questions
Nicole Bidoit, Melanie Herschel, Katerina Tzompanaki

To cite this version:
Nicole Bidoit, Melanie Herschel, Katerina Tzompanaki. Efficient computation of polynomial explanations of Why-Not questions. 31ème Conférence sur la Gestion de Données - Principes, Technologies et Applications - BDA 2015, Sep 2015, Île de Porquerolles, France. <hal-01182104>
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

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

ABSTRACT

Answering a Why-Not question consists in explaining why the result of a query does not contain some expected data, called missing answers. This paper [6] focuses on processing Why-Not questions following a query-based approach that identifies the culprit query components. The first contribution of this paper 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. Moreover, this formalism allows us to represent Why-Not explanations to extended relational models having for instance probabilistic or bag semantics. Computing the Why-Not explanation is a complex process and the second contribution of the paper is an algorithm that efficiently generates the aforementioned polynomials that answer Why-Not questions. An experimental evaluation demonstrates the practicality of the algorithm both in terms of efficiency and explanation quality, compared to existing algorithms.

Répondre à des questions de type "Pourquoi pas" (Why Not) consiste à expliquer pourquoi certaines données appelées réponses manquantes sont absentes du résultat d’une requête. Cet article traite de questions de type "Pourquoi pas" en suivant une approche "requête", c’est-à-dire que les explications sont fournies par les combinaisons de conditions de la requête qui sont responsables de la non obtention de certaines réponses. La première contribution est une définition générale de ce qu’est l’explication d’une question "Pourquoi pas" sous la forme d’un polynôme. Intuitively, ce polynôme fournit toutes les voies à explorer pour récupérer les réponses manquantes. De plus, cette définition permet, avec le même formalisme, de s’intéresser à des extensions du modèle relationnel tel que la sémantique multi-ensembliste ou probabiliste. La deuxième contribution de cet article est liée au calcul des explications d’une question "Pourquoi pas". Un algorithme efficace est présenté, accompagné d’une validation expérimentale et d’une étude comparative.

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 traveler) 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 very 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 as well as other types of queries like Top-k and 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 conditions of the query that are responsible for pruning data relevant to the missing answers. Existing methods producing query based explanations are not satisfactory, as they return different explanations for the same SQL query, and miss explanations. This is due to the fact that these algorithms are designed over query trees and thus, the explanations depend on the topology of a given tree and indeed to the ordering of the query operators in the query tree.

EXAMPLE 1.1. Consider 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 for the query. It also shows the tree operators that WhyNot [9] (⋆) and NedExplain [5] (⋆) return as query-based explanations as well as the tree operators returned as part of hybrid explanations by Conseil [18, 19] (⩾). 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.

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. To this problem, we make in this paper the following contributions:
SELECT airline, country FROM Airline A, Country C WHERE ccode = code AND year < 1985

<table>
<thead>
<tr>
<th>Airline</th>
<th>airline year</th>
<th>code</th>
<th>country</th>
<th>ccode</th>
</tr>
</thead>
<tbody>
<tr>
<td>KLM</td>
<td>1919</td>
<td>1</td>
<td>Australia</td>
<td>1</td>
</tr>
<tr>
<td>Qatar</td>
<td>1993</td>
<td>1</td>
<td>France</td>
<td>2</td>
</tr>
<tr>
<td>Aegean</td>
<td>1987</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Emirates</td>
<td>1985</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Example query and data

\[
\Pi_{\text{airline,country}}(\sigma_{\text{year}<1985} M_{\text{code}} \circ \sigma_{\text{year}=1985} \circ \bullet) \quad \Pi_{\text{airline,country}}(\sigma_{\text{year}=1985} \circ \bullet) \quad \text{Country} \quad \text{Airline}
\]

Figure 2: Reordered query trees for the query of Fig. 1 and algorithm results (Why-Not ⊂, NedExplain ⊂, Conseil •)

Extended formalization of Why-Not explanation polynomial.

In preliminary work [3, 4], we introduced polynomials as Why-Not explanations in the context of the relational model under set semantics. A polynomial provides a complete explanation and is independent of a specific query tree representation, solving the problems illustrated by the Ex. 1.1.

This paper significantly extends on the preliminary notion of a Why-Not explanation: the overall framework has been considerably simplified while the notion of Why-Not explanation is extended to be used in the context of relational model under set, 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.

Efficient Ted++ algorithm.

In our preliminary work [3, 4], we presented a naive algorithm computing Why-Not explanations. We show that its runtime complexity is impractical and propose a totally new algorithm, Ted++. Ted++ is capable of efficiently computing the Why-Not explanation polynomial, based on techniques like smart data partitioning (allowing for a distributed computation) and advantageous translation of expensive database evaluations by mathematical calculations.

Experimental validation.

We experimentally validate both the efficiency and the effectiveness of the solutions proposed in this paper. These experiments include a comparative evaluation of 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.

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 in detail the Ted++ algorithm in Sec 4. Finally, we present our experimental setup and evaluation in Sec. 5 before we conclude in Sec. 6.

2. RELATED WORK

Recently, we observe the trend that growing volumes of data are processed by programs developed not only by expert developers but also by less knowledgeable users (creation of mashups, use of web services, etc.). These trends have led to the necessity of providing algorithms and tools to better understand and verify the behavior and semantics of developed data transformations, and various solutions have been proposed so far, including data lineage [12] and more generally data provenance [11], (sub-query) result inspection and explanation [2, 15, 27], query conditions relaxation [25], transformation specification simplification [23, 26], etc.

The work presented in this paper falls in the category of data provenance research, and specifically explaining missing answers from query results. Due to the lack of space, the subsequent discussion focuses on this sub-problem, thus on algorithms answering Why-Not questions. Tab. 1 summarizes these algorithms, first classifying them according to the type of explanation they generate (instance-based, query-based, hybrid, ontology-based or refinement-based). The table further shows whether an algorithm supports simple Why-Not questions, i.e., questions where each condition impacts one relation only, or more complex ones. The last two columns summarize the form of a returned explanation and the class of query supported by an algorithm respectively.

### Query-based and hybrid explanations.

Why-Not [9] takes as input a simple Why-Not question and returns so-called picky query operators as query-based explanation. To determine these, the algorithm first identifies tuples in the source database that satisfy the conditions of the input Why-Not question and that are not part of the lineage [12] of any tuple in the query result. These tuples, named compatible tuples, are traced through the query operators of a query tree representation to identify which operators include them in their input but not in their output. In [9] the algorithm is shown to work for queries involving selection, projection, join, and union (SPJU query). NedExplain [5] is very similar to Why-Not in the sense that it supports simple Why-Not questions and returns a set of picky operators as query-based Why-Not explanation as well. However, it supports a broader range of queries, i.e., queries involving selection, projection, join, and aggregation (SPJIA queries) and unions thereof and the computation of picky operators is significantly different. In this work, we support a wider class of Why-Not questions (complex ones) and provide a new formalization of Why-Not explanation as polynomials.

Conseil [18, 19] 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 both the data to be possibly incomplete and the query to be possi-
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. Then, in Sec. 3.2, we define the explanation of a Why-Not question as a polynomial and in Sec. 3.3 we provide a unified general framework for Why-Not explanations in the context of probabilistic, set, and bag semantics databases.

3.1 Preliminaries

For the moment, let us consider only discussion to relational databases under set semantics. A database schema \( \mathcal{S} \) is a set of relation schemas. A relation schema \( R \) is a set of attributes. We assume each attribute of \( R \) qualified, i.e., of the form \( R.A \) and, for the sake of simplicity we assume a unique domain \( D \). \( \mathcal{I} \) denotes a database instance over \( \mathcal{S} \) and \( \mathcal{I}(R) \) denotes the instance of a relation \( R \in \mathcal{S} \). We assume that each database relation \( R \) has a special attribute \( R.Id \), which is used as identifier for the tuples in \( \mathcal{I}(R) \). For any object \( O \) (a relational schema, database schema, condition etc.), \( \mathcal{A}(O) \) denotes the set of attributes occurring in \( O \). Finally, a condition \( c \) over \( \mathcal{S} \) is defined as an expression of the form \( R.A \theta a \) where \( a \in D \), or of the form \( \neg (R.A \theta a) \), where \( \theta \in \{=, \neq, <, \leq, >, \geq \} \).

A condition over two relations is complex, otherwise it is simple. In this article, we consider conjunctive queries with inequalities. Note that, in our approach, the database schema \( \mathcal{S} \) denotes the query input schema. In an SQL-like approach, each time we need an instance of a relation, we refer to it by a different name. In this way, we are able to correctly define Why-Not questions in the case of self-join.

Definition 3.1 (Query Q). A query \( Q \) is specified by the triple \( (\mathcal{S}, \mathcal{I}, C) \), where \( \mathcal{S} \) is a database schema, \( \mathcal{I} \subseteq \mathcal{A}(S) \) is the projection attribute set, and \( C \) is a set of conditions over \( \mathcal{A}(S) \). The semantics of \( Q \) are given by the relational algebra expression

\[
\pi_{Q}(\mathcal{U}(\mathcal{I} \rightarrow \{x \mid x \in \mathcal{S}(R)\})].
\]

The result of \( Q \) over a database instance \( \mathcal{I} \) of \( \mathcal{S} \) is denoted by \( Q[\mathcal{I}] \). Note here that we are not concerned about the evaluation/optimization of \( Q \).

Example 3.1. Fig. 3 describes our running example. Fig. 3(a) displays an instance \( \mathcal{I} \) over \( \mathcal{S} = \{R, S, T\} \). Fig. 3(b) displays a query \( Q \) over \( \mathcal{S} \) whose conditions have been named for convenience. \( R.B=T.B \) and \( T.D=S.D \) are simple conditions. Moreover, \( Q[\mathcal{I}]=\{(R.B=5, S.D=4, T.C=9)\} \).

In our framework, a Why-Not question specifies missing tuples from the result of a query \( Q \) though a conjunctive set of conditions. As a Why-Not question is related to the result of \( Q \), the conditions of the Why-Not question are restricted to the attributes of the output schema of \( Q \).

Definition 3.2 (Why-Not question). A Why-Not question \( WN \) w.r.t. \( Q \) is defined as a set of conditions over \( \Gamma \).

The notion of complex and simple conditions is extended to complex and simple Why-Not questions in a straightforward manner. 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 these can be computed using \( WN \) as in Def. 3.3.

Definition 3.3 (Compatible tuples). Consider the query \( WN = \{\mathcal{S}, \mathcal{A}(S), WN\} \), where \( S \) is the input schema of \( Q \). The set \( CT \) of compatible tuples is the result of the query \( WN \) over \( \mathcal{I} \).
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 \( Q \) 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 \( \tau_1(CT) \cap Q[Z]=\emptyset \).

**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 is 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 \( Q_{WN} = \pi_{R.B,S.D,T.C} \ L \), which contains 12 tuples. For example, one compatible tuple is \( \tau_1 = (R.Id:1, R.A:1, R.B:3, S.Id:5, S.D:4, S.E:8, T.Id:8, T.B:3, T.C:4, T.D:5) \).

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

### 3.2 The Why-Not Explanation Polynomial

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 \( \tau \).

**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 \( \mathcal{E}_{\tau} = \{c \in C \text{ and } \tau \not\in c\} \).

**Example 3.3.** Reconsider the compatible tuple \( \tau_1 \) in Ex. 3.2. The conditions of \( Q \) (see Ex. 3.1), not satisfied by \( \tau_1 \) are: \( c_1, c_3 \), and \( c_4 \). So, the explanation for \( \tau_1 \) is \( \mathcal{E}_{\tau_1} = \{c_1, c_3, c_4\} \).

Having defined the explanation wrt one compatible tuple, the explanation for \( WN \) is obtained by simply summing up the explanations for all the compatible tuples in \( CT \). This leads to an expression of the form \( \sum_{\tau \in CT} \prod_{c \in \mathcal{E}_{\tau}} c \). 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 \( \tau \in CT \) implies 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

\[
\text{PEX} = \sum_{\varepsilon \in E} \text{coef}_E \prod_{c \in \varepsilon} c.
\]

where \( E = 2^C \), \( \text{coef}_E \in \{0, \ldots, |CT|\} \) is the number of tuples in \( CT \) sharing \( \varepsilon \) as an explanation, and \( \sum_{\varepsilon \in E} \text{coef}_E = |CT| \).

Intuitively, \( E \) contains all possible explanations, i.e., condition combinations and each of these explanations prunes from zero to at most \( |CT| \) compatible tuples. Each compatible tuple is pruned by exactly one condition combination, which is why the sum of all coefficients is equal to the number of compatible tuples.

We mentioned before that 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 as in Def. 3.6 offers through its coefficients some useful hints to users interested in the number of recoverable tuples. More precisely, by isolating an explanation \( \mathcal{E} \) to repair, we can obtain an upper bound for the number of compatible tuples that can be recovered. The upper bound is calculated as the sum of the coefficients of all the explanations that are sub-sets of (the set of conditions of) \( \mathcal{E} \), because when \( \mathcal{E} \) is changed it is likely that some sub-combinations are also repaired.

**Example 3.4.** In Ex. 3.3 we found the explanation \( \{c_1, c_3, c_4\} \), leading 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: \( 2 \cdot c_1 \cdot c_2 + 2 \cdot c_1 \cdot c_3 + c_2 \cdot c_4 + 2 \cdot c_1 \cdot c_2 \cdot c_3 + c_2 \cdot c_3 \cdot c_4 \). In the polynomial, each addend, composed by a coefficient and an explanation, captures a way to obtain missing tuples. For instance, the explanation \( c_1 \cdot c_2 \cdot c_3 \) indicates that we may recover some missing answers if \( c_1 \) and \( c_2 \) are changed and \( c_3 \) is fixed. Then, the sum of its coefficient 4 and the coefficient 2 of the explanation \( c_1 \cdot c_2 \cdot c_4 \{\{c_1, c_2\} \subseteq \{c_1, c_2, c_4\}\} \) indicates that we can recover from 0 to 6 tuples.

As the presentation of the polynomial per se may sometimes be cumbersome, and thus not easy for a user to manipulate, some post-processing steps could be applied. For example, depending on the application or needs, only a subset of the explanations could be returned like minimum explanations (i.e., for which no sub-explanations exist), or those explanations supposed to recover a specific number of tuples, or having 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 semantics and probabilistic databases.

\( K \)-relations, as introduced in [14], capture in a unified manner relations under probabilistic, bag or set semantics. Briefly, a \( K \)-relation maps tuples to elements of a set \( K \), that is \( K \)-relation tuples are annotated with elements in \( K \). In our case, we consider that \( K \) is a set of 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 \( \varepsilon \in Q(T) \) is modelled as the polynomial obtained by the positive algebra on \( K \)-relations, proposed in [14]. Briefly, each \( t \) is annotated with a polynomial whose variables are tuple identifiers and coefficients are natural numbers. Following [14]’s algebra, if \( \eta \) results from a selection operator on a tuple \( t_1 \) annotated with \( Id_1 \), then \( \eta \) 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 \( Id_1 \cdot Id_2 \).

We compute the generalized Why-Not explanation polynomial as follows. Firstly, we compute the how-provenance for compatible tuples in \( CT \) by evaluation of the query \( Q_{WN} \) (Def. 3.3) wrt the algebra in [14]. Recall that \( Q_{WN} \) contains only selection and join operators. Thus, we assume that each compatible tuple \( \tau \) in \( CT \) is annotated with its how-provenance polynomial, denoted by \( \eta_\tau \).
In a second step, 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 annotated with the expression $\eta_{\tau}E_{\tau}$.

Finally, as before, we sum the combined expressions for all compatible tuples. The result is the generalized Why-Not explanation $P_{\text{gen}} = \sum_{\tau \in CT} \eta_{\tau}E_{\tau}$.

We now briefly comment on how $P_{\text{gen}}$ is instantiated to deal either with the set, bag or probabilistic semantics. Indeed, the specialization of $P_{\text{gen}}$ relies on the interpretation of the elements in $K$, that is on a function $\text{Eval}$ from $\mathbb{K}$ to some set $L$. For the set semantics, each tuple in a relation occurs only once. This results in choosing $L$ to be the singleton $\{1\}$ and mapping each tuple identifier to 1. It is then quite obvious to note, for the set semantics, that $P_{\text{gen}} = PEX$ (Def. 3.6). In the same spirit, for bag semantics, $L$ 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 parameterized by the mapping $\text{Eval}$ 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

$$P_{\text{gen}} = \sum_{E \in \mathbb{E}} \sum_{\tau \in CT \text{ s.t. } \tau \cdot E = E} \text{Eval}(\eta_{\tau})E$$

where $E = \mathbb{E}_{\tau}, \eta_{\tau}$ is the how-provenance of $\tau$, and $\text{Eval}: K \rightarrow L$ evaluates the elements of $K$ to values in $L$.

The specializations of $P_{\text{gen}}$ share the same explanations (terms of the polynomial), capturing the same ‘erroneous’ parts of the query. However, the coefficients are interpreted wrt the how-provenance.

### 4. TED++ Algorithm

In [3], we have introduced $Ted$, a naive algorithm that implements the definitions of [3] for Why-Not explanations in a straightforward manner. Briefly, $Ted$ enumerates the set of compatible tuples $CT$ by executing the query $Q\text{WN}$ (Def. 3.3). Then, it computes all explanations for each compatible tuple in $CT$, which leads to the computation of the final Why-Not explanation. However, both of these steps make $Ted$ computationally prohibitive. Not only is the computation of $CT$ time and space consuming as it often requires cross product executions, but also the iteration over this (potentially very large) set is time consuming. $Ted$’s time complexity is $O(n^{|S|})$, $n=\max(|I_R|)$, $R \in S$. As experiments in Sec. 5 confirm, this complexity renders $Ted$ of no practical interest.

To overcome the poor performance of $Ted$, 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, the polynomial is computed based on mathematical calculations.

Theorem 4.1 states that $Ted++$ is sound and complete w.r.t. Def. 3.6.

**Algorithm 1: $Ted++$**

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

**Output:** $PEX$

1. $E \leftarrow \text{powerset}(C)$;
2. $P \leftarrow \text{validPartitioning}(S,WN)$ *(Def. 4.1)*
3. for $Part \in P$
4. \hspace{1cm} $CT_{|Part} \leftarrow (Part, A(\text{Part}), WN_{|Part})|\mathcal{I}_{|Part}|$;
5. coefficientEstimation($E, Part$);
6. $PEX \leftarrow \text{post-processing})$;
7. return $PEX$;

**Theorem 4.1.** Given a query $q$, a Why-Not question $WN$ and an input instance $I$, $Ted++$ computes exactly $PEX$.

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 introduced in the detailed discussions, we will rely on our running example, for which Fig. 4 shows all relevant intermediate results. It should be read bottom-up. For convenience, in our examples, we use subscript $i$ instead of $c_i$.

#### 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 $P = \{Part_1, \ldots, Part_k\}$, is valid if each $Part_i$, $i \in \{1, \ldots, k\}$ is minimal w.r.t. the following property:

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

The partitioning of $S$ allows for handling compatible tuples more efficiently, by ‘cutting’ them in distinct meaningful ‘chunks’, avoiding combining chunks over distinct partitions through cross product. We refer to the chunks of compatible tuples as partial compatible tuples and group them in sets depending on the partition they belong to. The set $CT_{|Part}$ of partial compatible tuples w.r.t $Part \in P$ is obtained by evaluating the query $Q_{|Part}=(Part, A(\text{Part}), WN_{|Part})$ over $\mathcal{I}_{|Part}$ (Alg. 1, line 4). $WN_{|Part}$ and $\mathcal{I}_{|Part}$ denote the restriction of $WN$ and $I$ over the relations in $Part$, respectively.

**Example 4.1.** The valid partitioning of $S$ is $Part_1=\{R, S\}$ (because of the condition $R.B<S.D$) and $Part_2=\{T\}$. The sets of partial compatible tuples $CT_{|Part_1}$ and $CT_{|Part_2}$ 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 individual $CT_{|Part_i}$.

**Lemma 4.1.** Let $P$ be the valid partitioning of $S$. Then, $CT = \bigcup_{Part \in P} CT_{|Part}$.

Indeed, Lemma 4.1 makes it clear how $CT$ is computed from partial compatible tuples. Our algorithm is designed in a way that avoids computing $CT$ and relies on the computation of $CT_{|Part_i}$ only.
Algorithm 2: coefficientEstimation

\begin{algorithmic}
\State \textbf{Input} $E$ explanations space, $P$ valid partitioning of $S$
\For {$E \in E \text{ access in ascending size order}$} \Do
\If {$|E| = 1$} \Then
\State materialize $V_E$;
\State $\beta_E \leftarrow \text{Eq. (B)}$;
\Else
\If {a subcombination of $E \neq 0$} \Then
\State $\{E1, E2\} \leftarrow \text{subCombinationsOf}(E)$;
\State $V_{E1} \leftarrow V_E \cap V_{E2}$; $\Gamma_2$ is the output schema of $V_{E1}$\;\If {$\Gamma_1 \neq \emptyset$} \Then
\State $V_E \leftarrow V_{E1} \bowtie \Gamma_2$; $\alpha_E \leftarrow \text{Eq. (A)}$;
\Else
\State $|V_E| = |V_{E1}| + |V_{E2}|$;
\State $\beta_E \leftarrow \prod_{E \in P \setminus \{\text{part}_E\}} |CT|_{\text{part}_E}|$; \EndIf
\Else
\State $|V_E| = |V_{E1}| + |V_{E2}|$;
\State $\beta_E \leftarrow \prod_{E \in P \setminus \{\text{part}_E\}} |CT|_{\text{part}_E}|$; \EndIf
\Else
\EndIf
\EndIf
\EndFor
\end{algorithmic}

Next, we compute the number of compatible tuples eliminated by each possible explanation, starting with the partial compatible tuple sets previously defined. These numbers approximate the coefficient of the explanations in the polynomial. Since from this point on, we are only handling (partial) compatible tuples, we omit the word ‘compatible’ to lighten the discussion.

### 4.2 Polynomial Coefficient Estimation

Each set $E$ in the powerset $\mathbf{E}$ is in fact a potential explanation that is further processed. In order to do that, we associate $E$ with (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 wrt $E$, and, (iii) the number $\alpha_E$ of tuples eliminated by $E$.

Alg. 2 describes how we process $E$ in ascending order of explanation size, in order to compute $\alpha_E$. Each step deals with explanations of size $s$, reusing results from previous steps avoiding cross product computations through mathematical calculations.

We first determine the set of partitions for an explanation $E$ as $\text{part}_E = \bigcup_{E \in E} \{\text{Part}_E\}$, where $\text{Part}_E$ 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, $E = \{c_1, c_2\}$ 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.

$$\alpha_E = \beta_E \prod_{\text{part}_E \in P \setminus \{\text{part}_E\}} |CT|_{\text{part}_E}|,$$

(A)

where $\text{part}_E = P \setminus \{\text{part}_E\}$. Note that when $\text{part}_E$ is empty, we abusively consider that $\prod_1 = 1$.

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

### Atomic explanations

We start with explanations $E$ containing only one condition $c$ (Algorithm 2 lines 3-5), which we call *atomic* explanations. To find the number of eliminated partial tuples $\beta_E$, we firstly compute the set of passing partial tuples w.r.t. $c$, which we store in the view $V_c$.

$$V_c = \bigg\{ \begin{array}{ll}
\pi_{(R,R\in \text{part}_1)\rightarrow (\sigma_1(CT|_{\text{part}_1}) \text{ if part}_E = \{\text{Part}_1\})} & \text{if part}_E = \{\text{Part}_1\} \\
\pi_{(R,R\in \text{part}_1\cup\text{part}_2)\rightarrow (|CT|_{\text{part}_1}|\cap |CT|_{\text{part}_2}|) \text{ if part}_E = \{\text{Part}_1, \text{Part}_2\}} & \text{if part}_E = \{\text{Part}_1, \text{Part}_2\} \\
\end{array} \bigg\}$$

Then, $\beta_E = \prod_{\text{part}_E \in P \setminus \{\text{part}_E\}} |CT|_{\text{part}_E}|$.

**Example 4.3.** For $c_2$, we have $\text{part}_{c_2} = \{\text{Part}_1, \text{Part}_2\}$, so $V_{c_2} = \pi_{(R,R\in \text{part}_1\cup\text{part}_2)\rightarrow (|CT|_{\text{part}_1}|\cap |CT|_{\text{part}_2}|)}$. This results in $|V_{c_2}| = 4$, and by Eq. (B) we obtain $\beta_{c_2} = |CT|_{\text{part}_1}| - |CT|_{\text{part}_2}| - |V_{c_2}| = 3 \times 4 = 12$. Since all partitions of $P$ are in $\text{part}_{c_2}$, applying Eq. (A) results in $\alpha_{c_2} = \beta_{c_2} = 8$. For $c_3$, $\beta_{c_3} = |CT|_{\text{part}_1}| - |V_{c_3}| = 4 - 2 = 2$, so $\alpha_{c_3} = 3\times2 = 6$. Fig. 4 (second level) displays the process for all atomic explanations.

### Non atomic explanations

Now, consider $E = \{c_1, \ldots, c_n\}$, $n \geq 1$ (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 $V_{c_i}$ for $i = 1, \ldots, n$ are well-defined. Firstly, we compute the view $V_E$ storing the passing partial tuples wrt $E$ as $V_E = V_{c_1} \cap \ldots \cap V_{c_n}$. 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., $\beta_E = |V_{c_1} \cap \ldots \cap V_{c_n}|$. By the well-known DeMorgan law [29], we have $\beta_E = |V_{c_1} \cup \ldots \cup V_{c_n}|$, which spares us from computing the complements of $V_{c_i}$.

To compute the cardinality of the union among $V_{c_i}$, we rely on the Principle of Inclusion and Exclusion for counting [16]:

$$\bigg| \bigcup_{i=1}^n V_{c_i} \bigg| = \sum_{J \subseteq \{1,\ldots,n\}} (-1)^{|J|+1} \bigg| \bigcap_{j \in J} V_{c_j} \bigg|$$

We further rewrite the previous formula to re-use results obtained for sub-combinations of $E$, obtaining Eq. (C).

$$\bigg| \bigcup_{i=1}^n V_{c_i} \bigg| = \sum_{i=1}^n V_{c_i} \bigg| + \bigg| \bigcap_{j \in J} V_{c_j} \bigg|$$

(C)

At this point, we have all the necessary data to compute $\beta_E$. However, so far we assumed that the conditions in $E$ have the same schema. In the general case, we have to “extend” the schema of a view $V$, to the one of $V_E$, in order to have well-defined set operations. The cardinality of an extended $V_E$ is given by Eq. (D).

$$\bigg| V_E^{\text{ext}} \bigg| = \prod_{\text{part}_E \in P \setminus \{\text{part}_E\}} |CT|_{\text{part}_E}| \bigg| V_E \bigg|$$

(D)

Based on Eq. (D) we obtain Eq. (E) that generalizes Eq. (C).
The number of eliminated tuples is then calculated by the complement of the set of incompatible tuples as the part 5 of the view schema, thus

\[ |V| \]

\[ |V| = |\{\text{compatible tuples}\}^c| \]

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

Using the above, we trivially compute the number \(\beta_E\) of eliminated partial tuples as the complement of \(|\{\cup_{i=1}^n V_{c_i}\}^{ext}\) (Alg. 2, line 17). The number of 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 \(c_3c_4\), Eq. (E) gives: \(|(V_2 \cup V_3)^{ext}| = |V_2^{ext}| + |V_3^{ext}| = 6 + 6 = 12\). The view \(V_3\) has already a matching schema, thus \(|V_3^{ext}| = 6\). For \(V_2\), \(\Gamma = \{T_1d\}\), we thus apply Eq. (D) and obtain \(|V_2^{ext}| = |V_{Part2}| = 6\). So, \(|V_2^{ext}| = 6\). For \(V_3\), \(|V_3^{ext}| = 6\), we remain to be calculated. Intuitively, because \(V_2\) and \(V_3\) target schemas share attribute \(T_1\), \(V_2 = V_3 \otimes T_1\). The view \(V_3\) is materialized and contains 2 tuples (as shown in Fig. 4). So, finally, from Eq. (E) we obtain \(|(V_2 \cup V_3)^{ext}| = 4 + 6 - 2 - 1 + 0 = 8\).

Now focus on the explanation \(c_3c_5\). The schemas of \(V_3\) and \(V_4\) are disjoint and intuitively \(V_3 \otimes V_4\). Here, \(\beta_3 = 0\). We materialize, we simply calculate \(|V_3| = 6\). Then, \(\beta_{35} = 12 - (12 + 6) = 0\). As we will see later, these steps are never performed in our algorithm. The fact that \(c_5\) does not eliminate any tuple (see \(c_3\) in Fig. 4) implies that neither do any of its super-combinations. Thus, a priori we know that \(\beta_{35} = 0\).

Finally, we illustrate the case of a bigger size combination, for example \(c_4c_3c_4\) of size 3. Eq. (E) yields \(|(V_2 \cup V_3 \cup V_4)^{ext}| = |(V_2 \cup V_3)^{ext}| + |V_4^{ext}| - |(V_2 \times V_3)^{ext}| - |(V_2 \times V_4)^{ext}| + |(V_3 \times V_4)^{ext}| + |V_2 \times V_3 \times V_4)^{ext}|. All terms of the right side of the equation are available from previous iterations, except for \(|(V_2 \times V_3 \times V_4)^{ext}|. As before, we check the common attributes of the views and obtain \(V_2 = V_2 \times R_{Id,S,I,d}^{T_1d}V_3\). So, \(|(V_2 \times V_3 \times V_4)^{ext}| = 2 - 2 + 1 - 0 = 1\). However, \(\beta_{34} = 12 - 9 = 3\). In the same way, we compute all the possible explanations until \(c_1c_2c_3c_4c_5\).

**View Materialization: when and how.**

To decide when and how to materialize the views for the explanations, we partition the set of the views associated with the conditions in \(E\). Consider the relation \(\sim\) defined over these views by \(V_i \sim V_j\) if the target schemas of \(V_i\) and \(V_j\) have at least one common attribute. Consider the transitive closure \(\sim^*\) of \(\sim\) and the induced partitioning of \(V_E\) through \(\sim^*\).

When this partitioning is a singleton, \(V_E\) needs to be materialized (Alg. 2, line 9). The materialization of \(V_E\) 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 $c3c4c5$, $V_{234}$ can either be computed through $V_{23} \times V_4$ or $V_{23} \times V_4$, or $V_{23} \times V_4$, or $V_{23} \times V_4$... because all these views are known from previous iterations. The choice of the query used to materialize $V_j$ is done based on a cost function. This function gives priority to materializing $V_j$ by means of one join, which is always possible: because $V_j$ 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 $V_{23} \times V_4$, or $V_{23} \times V_4$, or $V_{23} \times V_4$ is considered. In order to choose among the one-join queries computing $V_j$, we favor a one-join query $V_j \times V_j$ minimal w.r.t. $|V_j| + |V_j|$. For the example, and considering also Fig. 4 we find that $|V_2| + |V_4| = |V_4| + |V_3| = 5$ and $|V_3| + |V_4| = 3$. So, the query used for the materialization is $V_{34}$ (its result being empty in our example). Nevertheless, we avoid the materialization of $V_j$ if the partitioning is a singleton (Alg. 2, line 9 & 16), when for some sub-combination $E$ of $E$ it was computed that $o_E = 0$. In that case, we know an upper bound $o_E = 0$ (see Ex. 4.4).

If the partitioning is not a singleton, $V_j$ is not materialized (Alg. 2, line 14). For example, the partitioning for $c3c5$ is not a singleton and so the size $|V_{35}| = |V_3| \times |V_5| = 6$.

**Post-processing.**

In Alg. 2 we associated with each possible explanation $E$ the number of eliminated tuples $o_E$. However, recall that the calculation of this number so far counts any tuple eliminated by $E$, even though the same tuples may be eliminated by some other 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 $o_E$:

$$cof_E = o_E - \left( \sum_{E \subseteq E} cof_{E'} \right)\quad (F)$$

**Example 4.5.** Consider known $cof_{12345} = 2$ and $cof_{123} = 2$. We have found in Ex. 4.4 that $o_{234} = 4$. With Eq. (F), $cof_{1236} = 4 - 2 - 2 = 0$. In the same way $cof_{12} = 4 - 0 - 2 - 2 = 0$. The algorithm leads to the expected Why-Not explanation polynomial already provided in Ex. 3.4.

### 4.3 Complexity analysis.

In the pseudo-code for Ted++ provided in Alg. 1, we can see that Ted++ divides into the phases of (i) partitioning $S$, (ii) materializing a view 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 $|V_j|$ before applying equations to compute $o_E$. Overall, we consider that all mathematical computations are negligible so, the worst case complexities of steps (i) through (iv) are $O(|S| + |WN|) + O(|S|) + O(2^{|C|}(|S| + |C|)) + O(2^{|C|})$. For large enough queries, we can assume that $|S| + |C| < 2^{|C|}$, in which case the complexity simplifies to $O(2^{|C|})$.

Obviously, 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 $n = \max(|T_i|, R \in S)$. 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 $O(n^{|S|})$. This yields a combined complexity of $O(2^{|C|}n^{|S|})$. However, Ted++ in the general case (more than one induced partitions), has a tighter upper bound: $O(n^{2k_1} + n^{k_2} + \ldots + n^{k_EN})$, where $k_E = |part_E|$, for all combinations $E$ and $N = 2^{|C|}$. It is easy to see that $n^{2k_1} + n^{k_2} + \ldots + n^{k_EN} < 2^{|C|}n^{|S|}$, when there is more than one partition.

### 5. EXPERIMENTAL EVALUATION

This section presents an experimental evaluation of Ted++, using real and synthetic datasets. In Sec. 5.1, we compare Ted++ with the existing algorithms returning query-based explanations, i.e., with NedExplain [5] and Why-Not [9]. Sec. 5.2 studies the runtime of Ted++ with respect to various parameters that we vary in a controlled manner. We have implemented Ted, Ted++, NedExplain, and Why-Not in Java. We have implemented the experiments on a Mac Book Air, running MAC OS X 10.9.5 with 1.8 GHz Intel Core i5, 4GB memory, and 120GB SSD. We used PostgreSQL 9.3 as database system.

#### 5.1 Comparative Evaluation

The comparative evaluation to 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 the experiments in this section, we have used data from three databases named crime, indb, and gov. The crime database corresponds to the sample crime database of the Trio system (available at http://infolab.stanford.edu/trio/) and was previously used to evaluate Why-Not and NedExplain. The data describes crimes and involved persons (suspects and witnesses). The indb database contains real-world movie data from IMDB (http://www.imdb.com). Finally, the gov database contains information about US congressmen and financial activities (data from http://bioguide.congress.gov, http://usaspending.gov, and http://earmarks.omb.gov).

For each dataset, we have created a series of scenarios (crime1gov5 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. The queries have been designed to include queries with a small set of conditions (Q6) or a larger one (Q1, Q3, Q5, Q7), containing self-joins (Q3, Q4), having empty intermediate results (Q2), as well as containing inequalities (Q2, Q4, Q5, Q6).

#### 5.1.1 Why-Not Explanation Evaluation

In Tab 1 we report that the explanations returned by Why-Not...
and NedExplain consist of sets of query conditions, whereas Ted++ returns a polynomial of query conditions. For comparison purposes, we trivially map Ted++’s Why-Not explanation to sets of conditions, e.g., \(3 \times c_3 = c_4 + 2 \times c_3 + c_6\) maps to \(\{c_1, c_3\}, \{c_3, c_6\}\).

For conciseness, we abbreviate condition sets, e.g., to \(c_3, c_6\).

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 crim1. 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++. Then, consider crime8. NedExplain returns the join \(c_2\) (\(S\text{\textunderscore hair}, P\)) - Why-Not falsely does not produce any explanations in this case. Ted++ indicates that except for this join, the selection \(c_3\) (\(\sigma_{\text{name} = 'B'}(P)\)) for instance is also an explanation. From a developer’s perspective, selections are typically easier or more reasonable to change. So, having the complete set of explanations potentially provides the developer with useful alternatives.

For the second argument consider crime5. NedExplain returns \(c_1\) (\(C\text{\textunderscore sector} \cdot W\)). 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, when the Why-Not algorithm produces an explanation \(c_{29}\) that misses some parts. Then, in gov3 NedExplain and Why-Not both return \(c_2\). However, let us now assume the developer prefers to not change this condition. Keeping in mind that those algorithms’ answers may change when changing the query tree, she may start trying different trees to possibly obtain a Why-Not explanation without \(c_2\). Knowing the explanation of Ted++ prevents her from spending any effort on this, as it shows that all explanations contain \(c_2\) as a part.

By mapping the explanation of Ted++ to sets of explanations, we have let aside an important property: the coefficients of the polynomial. For example, for the complete Why-Not explanation polynomial of crime8 is \(23\times c_1 + 20 \times c_4 + 4 \times c_1 + 8 \times c_2\). Assume that the developer would like to recover at least five missing tuples, by changing as few conditions as possible. The polynomial suggests to change either \(c_3\) or \(c_2\): they both require one condition change and provide the possibility of obtaining up to 20 and 8 missing tuples, respectively. The coefficient of \(c_1\) being 4, does not make \(c_1\) a good candidate, whereas \(c_2\) requires two condition changes. Clearly, the results of NedExplain or Why-Not are not informative enough for such a discussion.

### 5.1.2 Runtime Evaluation

We now compare the performance w.r.t. runtime of Ted++ with the other algorithms.

**Ted++ vs. NedExplain and Why-Not.**

For this comparative evaluation, we again consider scenarios crim1 through gov5 of Tab. 3 as they involve simple Why-not questions, making them processable by all three algorithms. Fig. 5 summarizes the runtimes in logarithmic scale for each algorithm and scenario. We observe that the runtime of Ted++ is always comparable to the runtime of NedExplain and that in some cases, it is significantly faster than Why-Not. Why-Not traces compatible tuples based on tuple lineage stored in Trio. As already stated in [5, 9], this design choice slows down Why-Not performance. On the contrary, both NedExplain and Ted++ compute the compatible data more efficiently by issu-

![Figure 5: Runtimes for Ted++, Ted, NedExplain and Why-Not](image-url)
ing SQL statements to the database and further using the unique identifiers of the source tuples. 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 definition and implementation issue of both Why-Not and NedExplain, which explains the sometimes faster runtime of Ted++ is the fact that their input is potentially much larger as it includes the full database instance instead of the compatible data only. Clearly, this slows the tracing of compatible data through the query tree.

Let us see what happens when Ted++ is slower than - but still comparable to - NedExplain, for example in gov1-gov3. 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., gov1).

Fig. 5 also reports runtimes for Ted on 6 out of 15 scenarios (for the others, Ted runs out of time). To experimentally demonstrate where Ted’s problem lies, we compare the time distribution in Ted and Ted++.

Fig. 6 divides the runtime into four common phases. Among these, the coefficientEstimation phase is the one that is inherently different in both algorithms. Ted iterates over the set of compatible tuples and computes the explanation for each one. Ted++ explores the search space of possible explanations and calculates, based on the number of passing partial compatible tuples, the number of compatible tuples eliminated by each explanation. Thus, this is the phase in which we expect to have an important runtime difference between Ted and Ted++. In reporting the phase-wise runtime, Fig. 6 cuts the bar for Ted in the scenarios crime7, gov1, gov2 and gov3 as the execution time is much higher compared to the other scenarios and to the runtime of Ted++ (the runtime of the coefficientEstimation phase is the label on the respective bars).

As said in Sec. 4, Ted’s main issue w.r.t. efficiency is its strong dependence on the number of compatible tuples. This is experimentally observed in Fig. 6: with the growth of the set of compatible tuples in the scenarios, the time dedicated to coefficientEstimation also grows (the scenarios are reported in an ascending order). Ted++ depends on the number of compatible tuples as well, but not as strongly as Ted. This can be seen in crime8 and crime7, or gov3 and gov1; while the number of tuples grows, Ted++’s coefficientEstimation phase 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. In addition to the number of compatible tuples, another important factor is the selectivity of the query conditions over the compatible data (i).

Experimental Setup.

For the parameter variations (i) and (ii), we use again the crime, imdb, and gov 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/). More specifically, we generate instances of 1GB and 10GB and further produce 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 complex 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 summarized in Tab. 2 (Q1-Qtpch) and the scenarios in Tab. 3 (crime1-crime4).

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 this series of queries is shown in Fig. 7 (a). Similarly, starting from query Q1, we introduce step by step complex conditions, yielding Q1-Q4. 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 in the coefficientEstimation phase, a view contains more tuples (=passing partial tuples) when the condition is less selective. This results in more computations in the super-combinations iterations, leaving space for further opti-
Adjusting the Why-Not question.

Next, we vary the type and the number of conditions in the Why-Not question WN. Fig. 8 shows the cases when we start (a) with a simple WN and progressively add more simple conditions and (b) start with a complex WN and progressively add more complex conditions.

The scenarios considered for Fig. 8 (a) have as starting point the simple scenario crime5 (see Tab. 3). Then, keeping the same input instance and query, we add attribute-constant comparisons to WN, a procedure 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 crime5 to crime5_c3). From this point, the runtime is dominated by the time to communicate with the database that has constant over all scenarios.

As we introduce complex conditions in the WN, 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_cc2, imdb_cc3). The number of partitions entailed by imdb_cc2, imdb_cc3, and imdb_cc4 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.

The last parameter we study is the input database size. To this end, we have created two scenarios, one with a simple and one with a complex Why-Not question WN, and both using the same query Q_{10}. We run both scenarios for database sizes 10MB, 100MB, 1GB, and 10GB. 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 informative, 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 and that the obtained runtimes match the theoretical expectations. Finally, the fact that the experiments were conducted on an ordinary laptop, with no special capabilities in memory or disk space, 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 it significantly reduces 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 saw that Ted++ scales well with various parameters, making it a practical solution. The proposed Why-Not explanation polynomial are easy to extend for unions of conjunctive queries, whereas an extension is not trivial for aggregation queries and 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 rewritings with minimum condition changes (i.e., small combinations), minimum side-effects (i.e., small coefficients), etc.

7. REFERENCES