Efficiently and Effectively Answering Why-Not Questions based on Provenance Polynomials

Nicole Bidoit, Melanie Herschel, Katerina Tzompanaki

Efficiently and Effectively Answering Why-Not Questions based on Provenance Polynomials

Nicole Bidoit, Melanie Herschel, Katerina Tzompanaki

RESEARCH REPORT

N° 8697

March 2015

Project-Teams OAK

ISSN 0249-6399   ISRN INRIA/RR--8697--FR+ENG

Efficiently and Effectively Answering Why-Not Questions based on Provenance Polynomials

Nicole Bidoit, Melanie Herschel *, Katerina Tzompanaki

Project-Teams OAK

Research Report n° 8697 — March 2015 — 25 pages

Abstract: The problem of answering Why-Not questions consists in explaining why the result of a query does *not* contain some expected data, i.e., *missing answers*. To solve this problem, we resort to identifying where in the *query*, data relevant to the missing answer were lost. Existing algorithms producing such *query-based explanations* rely on a query tree representation, potentially leading to different or partial explanations. This significantly impairs on the *effectiveness* of computed explanations. Here we present an effective, *query-tree independent* representation of query-based explanations, for a wide class of Why-Not questions, based on provenance *polynomials*. We further describe an algorithm that *efficiently* computes the complete set of these explanations. An experimental evaluation validates our statements.

Key-words: Why-Not questions, data provenance

* University of Stuttgart

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Répondre efficacement et pertinement à des requêtes Why-Not par des polynômes de provenance

Résumé : Une question de type "pourquoi pas" (Why Not) exprime une interrogation relative à l'absence dans le résultat d'une requête de certaines réponses attendues par l'utilisateur. Donc répondre à des questions de type "pourquoi pas" consiste à fournir une explication relative à l'absence de réponses. La solution que nous proposons cherche à identifier les éléments de la requête responsables de la perte de données ayant pu potentiellement contribuer à construction de ces réponses attendues mais manquantes. Les algorithmes existants qui produisent ce type d'explication dite "explication par la requête" sont développés en s'appuyant sur une représentation de la requête par un arbre. Cette approche a pour conséquence de produire des explications qui sont partielles d'une part et qui dépendent de l'arbre de requête choisi d'autre part. Celle-ci nuit donc à la qualité de l'explication. Dans cet article, nous proposons une méthode qui résoud, pour une classe de requêtes très grande, le défaut des travaux antérieurs en produisant des explications sous forme de polynômes de conditions inspirée par les polynômes de provenance. Un algorithme efficace est développé qui permet de calculer ces explications. La méthode est validée par cet algorithme et des expérimentations pertinentes.

Mots-clés : questions Why-Not, provenance de données

```
SELECT island,
       archipel
FROM Island I,
     Archipelago A
WHERE I.AID = A.AID
AND pop <= 20M
```

Island

island	pop	AID
$Maui$	$144K$	1
$Hawaii$	$3187K$	1
$Reunion$	$841K$	2
$Madagascar$	$22M$	NULL

Archipelago

AID	archipel
1	$Hawaii$
2	$Mascarene$

Figure 1: Example query and data

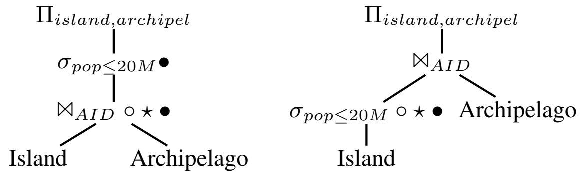


Figure 2: Reordered query trees for the query of Fig. 1 and algorithm results (Why-Not ∘, NedExplain ⋆, Conseil •)

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 query and data of Fig. 1, a developer (or scientist) may wonder why the island of Madagascar is missing from the result, even though she expected it to be part of it. 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 development, query debugging, query refinement, or what-if analysis. To help developers *explain missing answers*, different algorithms have recently been proposed for relational and SQL queries [5, 7, 14, 16, 17] as well as other types of queries (top-k [13], reverse skyline queries [19]). In this paper, we focus on relational queries, for which existing algorithms explain a missing answer either based on the data (instance-based explanations), the query (query-based explanations), or both (hybrid explanations). Moreover, we focus on solutions producing query-based explanations, as these are generally more efficient while providing sufficient information for query analysis and debugging. Taking a closer look at existing methods, we notice that these return different explanations for the same SQL query. This is due to the fact that these algorithms are designed over query trees rather than over the query, and thus trace data relevant to the missing answer, i.e., *compatible data* in a bottom-up manner through a specific query tree from which so-called *picky* operators are identified.

Example 1.1 *Consider the SQL query q and data $\mathcal{I}$ of Fig. 1 and assume that a developer wants an explanation for the absence of island Madagascar in the query result $q(\mathcal{I})$. So here, the why-not question is* "Why is tuple (island:Madagascar, archipel:x) not in $q(\mathcal{I})$?". *Fig. 2 shows two possible query trees for q. It also shows the picky operators that Why-Not [7] (∘) and NedExplain [5] (⋆) return as query-based explanations as well as query operators returned as part of hybrid explanations by Conseil [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 compatible tuple $(Madagascar, 22M, \texttt{NULL})$ from table $Island$ and this tuple does not find any join partner in table Archipelago.*

The above example clearly shows that existing algorithms have limited effectiveness when it comes to explaining missing answers. 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. To provide a more informative Why-Not answer we present in this paper the Why-Not answer in form of a polynomial. We then discuss both a naive and an efficient algorithm to compute this answer. Thus, the overall contribution of this paper is both an efficient and effective way to answer Why-Not questions for relational queries using provenance polynomials. In detail, our contributions are[1]:

Why-Not answer polynomial. Our formal framework supports a larger class of Why-Not questions w.r.t. previous works. The form of the Why-Not answer is unprecedented, as this paper is the first to formalize provenance polynomials providing fine-grained query based explanations. Intuitively, each addend of a polynomial represents one combination of the query conditions that *simultaneously* explain the missing answers and the set of all addends covers all possible such combinations. Moreover, the Why-Not answer is independent of the query tree representation of a query q. More precisely, all query trees which are equivalent to a conjunctive query (possibly) containing inequalities and which are obtained from each other by reordering of the operators have the same Why-Not answer polynomial up to isomorphism.

Naive *Ted* algorithm and efficient *Ted*++ algorithm. We present the *Ted* algorithm that correctly computes the Why-Not answer polynomial for a given query and a given Why-Not question. However, we show that its runtime complexity is impractical. Thus, we subsequently present an improved algorithm, *Ted*++, that is capable of efficiently computing the same Why-Not answer polynomial.

Experimental validation. We validate both the efficiency and the effectiveness of the solutions proposed in this paper through a series of experiments. These 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.

The remainder of this paper is structured as follows. Sec. 2 covers related work. Sec. 3 defines in detail our problem setting and the novel Why-Not answer polynomials. We briefly cover the naive *Ted* algorithm in Sec. 4 before we discuss in more detail the efficient *Ted*++ algorithm in Sec 5. We present our experimental setup and evaluation in Sec. 6 before we conclude in Sec. 7.

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 knowledgable 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 [9] and more generally data provenance [8], (sub-query) result inspection and explanation [11, 25], query conditions relaxation [23], visualization [10], or transformation specification simplification [20, 24]. The work presented in this paper falls in the category of data provenance research, focusing on a specific sub-problem that aims at explaining missing answers from query results. This sub-problem alone finds applications in various domains, e.g., information extraction [17], query debugging [15], distributed systems debugging [27], or image retrieval [3].

Due to the lack of space, the subsequent discussion focuses on algorithms proposed for answering *Why-Not questions*. Tab. 1 summarizes these approaches, first classifying them according to the type of explanation they generate (instance-based, query-based, hybrid, or modification-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 queries an algorithm supports, respectively.

[1] In a four-page workshop paper [4], we introduced the Why-Not answer polynomial as well as the naive *Ted* algorithm. Opposed to the workshop paper, the definitions here are more concise and additional theorems have been added. We also briefly summarize *Ted* here to clearly show that it is impractical, however, the focus of this paper clearly lies on the presentation of *Ted*++. Finally, the workshop paper does not include any experiments.

Table 1: Algorithms for answering Why-Not questions

Algorithm	**Why-Not question**	**Explanation format**	**Query**
		Instance-based explanations	
MA [17]	simple	source table edits	SPJ
Artemis [16]	complex	source table edits	SPJUA
Causality [22]	simple	causes (tuples) and responsibility	conjunctive queries
DL-Lite [6]	simple	additions to ABox	instance & conjunctive queries over DL-Lite ontology
		Query-based explanations	
Why-Not [7]	simple	query operators	SPJU
NedExplain [5]	simple	query operators	SPJUA
Ted [4]/Ted++	**complex**	**polynomial**	**conj. queries with inequalities**
		Hybrid explanations	
Conseil [14]	simple	source table edits + query operators	SPJAN
		Modification-based explanations	
ConQueR [26]	complex	rewritten query	SPJA
Top-k [13]	simple	rewritten query	top-k query
Skyline [19]	simple	rewritten query & why-not question	Reverse skyline query

Instance-based explanations. Both Missing-Answers (MA) [17] and Artemis [16] compute instance-based explanations in the form of source table edits (insertions or updates) that would be necessary to obtain the missing answers in the query result. Whereas MA returns correct explanations for simple Why-Not questions and SQL queries involving selection, projection, and join only (SPJ queries), Artemis supports complex why-not questions on a larger fraction of SQL queries (including union or aggregation, denoted SPJUA). Causality [22] theoretically studies the unification of instance-based explanations of missing answers and of data *present* in a query result, leveraging the concepts of causality and responsibility. The results apply to conjunctive queries. Finally, DL-Lite [6] leverages abductive reasoning and theoretically examines the problem of computing instance-based explanations for a class of simple Why-Not questions on data represented by a DL-Lite ontology. Here, the instance-based explanation consists in additions to the ontology's ABox (insertions to the instance data).

Query-based and hybrid explanations. Why-Not [7] 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 [9] 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 [7] 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 answer as well. However, it supports a broader range of queries, i.e., queries involving selection, projection, join, and aggregation (SPJA queries) and unions thereof and the computation of picky operators is significantly different. First, it does not restrict compatible tuples to source tuples not in the lineage of any result tuple. Second, based on a novel formal definition of query-based explanations, NedExplain computes a generally wider and detailed set of explanations than Why-Not.

Conseil [14] produces hybrid explanations that include an instance-based component (source table edits) 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 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. In general, this results in Conseil returning multiple hybrid explanations where the sets of picky operators of each explanation are different from those determined by Why-Not or NedExplain.

For comparison purposes, Tab. 1 also includes the naive *Ted* algorithm (previously introduced in a short workshop paper [4]) and *Ted*++. We observe that it is the first algorithm that computes query-based explanations for complex Why-Not questions, along with the novel format of Why-Not answer polynomial. Finally, *Ted++* applies on a different query fragment than the two previous algorithms, i.e., conjunctive queries with inequalities.

Modification-based explanations. Given a set of missing answers, an SPJUA query, and a source database, ConQueR [26] rewrites the query such that all missing answers become part of the output. Beyond SQL queries, the Top-K algorithm [13] focuses on changing k or preference weights to make the missing answer appear in the query result of a top-k query. Skyline [19] presents a solution for answering Why-Not questions in reverse skyline queries that modifies not only the query, but also the Why-Not question itself. Although these approaches are very interesting and valuable for the general purpose of *query refinement*, they are out of the scope of this paper.

3 Why-Not answers as Polynomial

We assume the reader is familiar with the relational model [1]. We briefly revisit certain notions in our context in Sec. 3.1. Why-Not questions are defined in Sec. 3.2. Sec. 3.3 extends the notion of compatible data introduced in previous work. Finally, we define the answer of a Why-Not question in Sec. 3.4 and discuss interesting properties in Sec. 3.5.

3.1 Preliminaries

We assume that a database schema $\mathcal{S}$ is a set of relation schemas. The set of attributes of a relation R always includes a special attribute R_Id because we assume that each tuple in an instance of R is referred to by an identifier Id. We denote by $Att(R)$ the set of attributes of R, except R_Id. We assume each attribute of R to be qualified, i.e., of the form $R.A$. For an instance $\mathcal{I}$ over S, we write $\mathcal{I}_{|R}$ for the component of $\mathcal{I}$ over R. We also assume available a set Var of variables $x, y, z, \ldots$ A condition c is either of the form $x\theta y$ or $x\theta a$, where $x, y{\in}Var$, a is a constant in a unique domain $\mathcal{D}$ and $\theta{\in}\{=, \neq, <, \leq\}$. A v-tuple v over R is a tuple of pairwise distinct variables over $Att(R)$. Note that a v-tuple does not associate a variable with the attribute R_Id. Next, $var(\cdot)$ is used to retrieve the set of variables from a structure, e.g., $var((x_1, \ldots, x_n))$ returns $\{x_1, \ldots, x_n\}$. The paper uses a notion of queries close to that of tableaux [2] and of inequality queries [21].

Definition 3.1 (Query tableau) *A query tableau (or simply query) q over schema $\mathcal{S}_q$ is a triple (s_q, T_q, C_q) where (i) the summary s_q is a set of distinguished variables with $s_q{\subseteq}var(T_q)$, (ii) the query skeleton T_q is a mapping associating one v-tuple v to each $R{\in}\mathcal{S}_q$ such that $var(T_q(R)){\cap}var(T_q(T){=}\emptyset$ for any distinct pair $R, T{\in}\mathcal{S}_q$, and (iii) the query condition C_q is a set of conditions over $var(T_q)$.*

To denote the result of q over $\mathcal{I}$, we use $q(\mathcal{I})$. Note also that our query definition does not allow expressing conditions involving the special attribute R_Id. These attributes thus do not appear in the tableau representation. Next, if a condition c in C_q refers via its variables to two distinct relations, we say that c is a *complex* condition, otherwise c is a *simple* condition.

Example 3.1 *Consider the database schema $\mathcal{S}_q{=}\{R, S, T\}$. Fig. 3 displays an instance $\mathcal{I}$ of $\mathcal{S}_q$ and the tableau representing a query q. The distinguished variables for q are underlined and the query condition is given in a special column. This query q corresponds to the following relational query:*

$$\pi_{R.B,S.D,T.C}(\sigma_{R.A>3}[R] \bowtie_B \sigma_{T.C\geq 8}[T] \bowtie_D \sigma_{S.E\geq 3}[S])$$

The condition $x_2{=}x_6$ is complex because the variables x_2, resp. x_6 refer to R, resp. T. The condition $x_4{=}x_8$ is complex as well. All other conditions are simple ones.

3.2 The Why-Not Question

Given a quey q, a Why-Not question is in general formulated as a predicate that is a disjunction of conditional tuples (c-tuples) [18]. Next, w.l.o.g., we concentrate on predicates composed of a single c-tuple. A full definition is available in [5]. The method presented here trivially extends to general predicates, but we omit a discussion due to space limitation.

Definition 3.2 (Why-Not question) *Let q=(s_q, T_q, C_q) be a query over $\mathcal{S}_q$. A (simple) Why-Not question is specified by a c-tuple t_c=$(t_v, \bigwedge_{i=1}^{n} c_i)$, where (i) t_v is a tuple of variables such that $var(t_v) \subseteq var(s_q)$, and (ii) c_i is a condition over the variables $var(t_v)$. Next, $t_c.cond$ denotes $\bigwedge_{i=1}^{n} c_i$. A Why-Not question t_c is complex if one of its condition is complex, ortherwise it is simple.*

Example 3.2 *Given the scenario of Ex. 3.1, we wonder why, in the answer $q(\mathcal{I})$, there is no tuple s.t. the value on $R.B$ is smaller than the one on $S.D$ and at the same time its value on $T.C$ is smaller or equal to 9. This Why-Not question is expressed by t_c=$((x_2, x_4, x_7), (x_2{<}x_4 \wedge x_7{\leq}9))$. In $t_c.cond$, $x_7{\leq}9$ is a simple condition whereas $x_2{<}x_4$ is a complex one. Consequently, t_c is a complex c-tuple.*

3.3 Compatible Data

Intuitively, compatible data designates any source tuples that potentially provide data to build the missing answer specified by t_c. The first step towards answering the Why-Not question consists in identifying, in the input instance $\mathcal{I}$, these tuples and more specifically combinations of them (called concatenated tuples) that would produce the missing answer in the absence of the restrictions of q. In a second step, discussed in the next section, we will identify the conditions in q that prune these concatenated tuples.

Example 3.3 *One can note in $t_c.cond$ of Ex. 3.2, that a missing answer depends on those tuples $t_R{\in}\mathcal{I}_{|R}$, $t_S{\in}\mathcal{I}_{|S}$ and $t_T{\in}\mathcal{I}_{|T}$ that satisfy $t_R(R.B){<}t_S(S.D)$ and $t_T(T.C){\leq}9$. Due to the complex condition, t_R and t_S need to be chosen in correlation with one another, whereas it is not the case for t_T. Thus, here the compatible concatenated tuples correlated with (t_R, t_S) are $(Id_1 Id_5)$, $(Id_1 Id_6)$ and $(Id_2 Id_6)$, while for t_T, each tuple in S, i.e., $Id_8, \ldots, Id_{11}$, is a compatible concatenated tuple.*

Previous approaches [5, 7] generate compatible tuples independently from each other, e.g., both Id_1 and Id_2 are considered compatible for t_R. However, Id_2 should not be considered compatible when Id_5 is chosen for t_S, which is not addressed by previous work. Therefore, we introduce *compatibility* on *concatenated tuples* rather than on single tuples. According to our definition, each compatible concatenated tuple (cc-tuple) would produce a missing answer tuple if it was not pruned by some condition(s) of the query.

Mappings. For a concise presentation, we need the functions defined and illustrated in Tab. 2. Function h_{Att} is extended to apply on the tableau and the c-tuple conditions respectively, whereas function $full$ naturally extends to cc-tuples, e.g., $full(Id_1 Id_5)$=$(R.A{:}1, R.B{:}3, S.C{:}1, S.D{:}4, S.E{:}8)$.

R

A	B	R_Id
1	3	Id_1
2	4	Id_2
4	5	Id_3
8	9	Id_4

S

C	D	E	S_Id
1	4	8	Id_5
3	5	3	Id_6
3	3	9	Id_7

T

B	C	D	T_Id
3	4	5	Id_8
3	8	1	Id_9
5	3	3	Id_{10}
5	9	4	Id_{11}

(a) Sample database instance $\mathcal{I}$

	R.A	R.B	S.C	S.D	S.E	T.B	T.C	T.D	C_q
R	x_1	$\underline{x_2}$							$x_1{>}3$ $x_2{=}x_6$
S			x_3	$\underline{x_4}$	x_5				$x_4{=}x_8$ $x_5{\geq}3$
T						x_6	$\underline{x_7}$	x_8	$x_7{\geq}8$

(b) A query tableau q

Figure 3: Sample instance (a) and query (b)

Table 2: Mapping functions

Function	Purpose	Example
h_{Att}:$Att(\mathcal{S}_q) \to var(T_{\mathcal{S}_q})$	Maps attribute names to variables in T_q.	$h_{Att}(R.A)$=x_1 $h^{-1}_{Att}(x_1) = R.A$
$full : ID \to \mathcal{I}$	Maps an identifier to its 'full' tuple.	$full(Id_1)$= $(R.A$:1, $R.B$:4)

Table 3: Compatibility tableau T_{t_c}

		R.A	R.B	S.C	S.D	S.E	T.B	T.C	T.D	$t_c.cond$
$Part_1$	R	x_1	x_2							$x_2 < x_4$
	S			x_3	x_4	x_5				
$Part_2$	T						x_6	x_7	x_8	$x_7 \leq 9$

Compatible concatenated tuples (cc-tuples). We are now ready to define the cc-tuples for the Why-Not question t_c. To this end, we consider the *compatibility* query T_{t_c}=(_, $T_q, t_c.cond$). Here we do not really care about the summary and thus omit it. Hence, in the following, T_{t_c} is specified by $(T_q, t_c.cond)$. Tab. 3 shows the tableau representation of T_{t_c} for our running example (ignore the grouping of rows for now). Intuitivelly, T_{t_c} captures the pattern that a cc-tuple should match and is formally defined as follows:

Definition 3.3 (cc-tuple w.r.t. t_c) *Let q be a query, t_c a Why-Not question, and $\mathcal{I}$ be an instance over $\mathcal{S}_q$=$\{R_1, \ldots, R_n\}$. The tuple τ=$(Id_1 \ldots Id_n)$, where $Id_i$$\in$$\pi_{R_Id}(\mathcal{I}_{|R_i})$, $\forall i$$\in$$[1, n]$ is a compatible concatenated tuple (cc-tuple) w.r.t. t_c if $full(\tau)$$\models$$h^{-1}_{Att}(cond)$. The set of cc-tuples w.r.t. t_c given $\mathcal{I}$ is denoted by $CCT(t_c, \mathcal{I})$.*

Example 3.4 *For τ=$(Id_1 Id_5 Id_8)$, it is immediate to check that $full(\tau)$$\models$$h^{-1}_{Att}(cond)$. This entails that τ is a cc-tuple w.r.t. t_c. In total, for our running example, we find 12 cc-tuples .*

3.4 The Why-Not Answer

Given the set $CCT(t_c, \mathcal{I})$ of cc-tuples, we define the Why-Not answer of t_c again relying on the skeleton T_q.

Definition 3.4 (cc-tuple tableau) *With the same assumption as above, given a cc-tuple τ w.r.t. t_c, the tableau T_τ associated with τ is defined by (_, T_q, $cond_\tau$$\cup$$C_q$), where $cond_\tau$ is the set of conditions over $var(T_q)$ induced by $full(\tau)$.*

Example 3.5 *Tab. 4 shows the tableau associated with the cc-tuple τ_1=$(Id_1 Id_5 Id_8)$. The condition sets $cond_\tau$ and C_q are displayed in two different columns.*

Let us now illustrate how T_τ is used to identify *picky* conditions in the query q (elements of C_q) that are considered responsible for pruning the cc-tuple τ from the query result.

Example 3.6 *First, we focus on τ_1 and on the two columns $cond_\tau$ and C_q of Tab. 4. The condition x_1=1 in $cond_\tau$ contradicts the condition x_1>3 of C_q. This leads us to conclude that x_1>3 is a picky condition. The conditions involving x_2 in $cond_\tau$ and C_q are simultaneously satisfied, as x_2=3 $\wedge$ x_6=3 $\wedge$ x_2=x_6 is true.*

Similarly, we identify the rest of the picky conditions in the column C_q and eventually obtain the set of picky conditions w.r.t. τ_1 that is $\{x_1$>3, $x_7$$\geq$8, x_4=$x_8\}$. This set provides all the conditions that have to be corrected so that the cc-tuple τ_1 appears in the result of q. We also say that τ_1 is a picked cc-tuple w.r.t. the conditions $\{x_1$>3, $x_7$$\geq$8, x_4=$x_8\}$.

Definition 3.5 (Picky conditions w.r.t. τ) *With the same assumptions as before, the set of picky conditions w.r.t. τ is defined by PO_τ=$\{c | c$$\in$$C_q$ and $cond_\tau \not\models c\}$.*

Table 4: cc-tuple tableau T_{τ_1}

	R.A	R.B	S.C	S.D	S.E	T.B	T.C	T.D	$cond_\tau$	C_q
R	x_1	x_2							$x_1{=}1$ $x_2{=}3$	$x_1{>}3, x_2{=}x_6$
S			x_3	x_4	x_5				$x_3{=}1$ $x_4{=}4, x_5{=}8$	$x_4{=}x_8$ $x_5 \geq 3$
T						x_6	x_7	x_8	$x_6{=}3$ $x_7{=}4$ $x_8{=}5$	$x_7 \geq 8$

Notation 3.1 *(Picked and passing cc-tuple τ w.r.t. op) A cc-tuple τ is* picked *w.r.t. a condition c iff $c{\in}PO_\tau$. Otherwise, τ is said to be a* passing *cc-tuple.*

The Why-Not answer includes an explanation for each cc-tuple $\tau{\in}CCT(t_c, \mathcal{I})$ and takes the form of a polynomial over conditions occuring in the query.

Definition 3.6 (Why-Not answer) *With the previous assumptions, the Why-Not answer is defined as*

$$TWNA(q, t_c, \mathcal{I}) = \sum_{\tau \in CCT(t_c, \mathcal{I})} \prod_{c \in PO_\tau} c$$

Example 3.7 *For the purpose of the presentation, we need to name each condition of C_q for our running example as follows:*

name	op_2	op_3	op_4	op_5	op_6
condition	$x_1{>}3$	$x_2{=}x_6$	$x_7{\geq}8$	$x_4{=}x_8$	$x_5{\geq}3$

In Ex. 3.6, we found that $\{op_2, op_4, op_5\}$ are the picky conditions for τ_1, which leads to the term $op_2 * op_4 * op_5$. Given the 12 cc-tuples of our example, we obtain the following polynomial: $2*op_2*op_5 + 2*op_2*op_4*op_5 + 4*op_2*op_3*op_5 + 2*op_2*op_3*op_4 + 2*op_2*op_3*op_4*op_5$. In the polynomial, each addend, composed by a coefficient and a condition combination, captures a way to obtain the missing answers. For instance, the combination $op_2*op_3*op_5$ indicates that if op_2 and op_3 and op_5 are correctly repaired, the missing answer will be produced by the query q. Then, the sum of its coefficient 4 and the coefficient 2 of its sub-combination op_2*op_5 indicates that we will get at most 6 instances of the missing answer by repairing this combination.

We justify modeling each PO_τ with a product by the fact that in order for τ to 'survive' up to the query result, every single picky condition w.r.t. τ must be 'repaired'. The sum of the products of each $\tau{\in}CCT(t_c, \mathcal{I})$ stems from the fact that, if any addend is 'correctly repaired', the associated τ will return the missing answer.

The coefficients of the polynomial provide the means to estimate the cardinality of the instances of the missing answers that will be obtained, when a combination x is repaired. More precisely, the sum of the coefficients of all sub-combinations of x provides an upper bound on the number of missing answer instances that could be recovered.

3.5 Why-Not Answer Properties

In this section, we compare the notion of Why-Not answer introduced in this paper (next called TED Why-Not answer) with the NedExplain Why-Not answer [5]. First, we show that the TED Why-Not answer is robust for a large class of trees. Then we show that for simple Why-Not question s, TED subsumes NedExplain.

Robustness of TED Why-Not answer. NedExplain Why-Not answers are defined for query trees, so we have to explain how TED Why-Not answers are defined for query trees. To this end, we associate a tableau query to a query tree in the obvious manner. To simplify the discussion, we assume that query trees are built using (i) relation schemas as leaf nodes and (ii) cartesian product $\times$ and selection σ_c where c is a condition as internal nodes. W.l.o.g., we do not consider projection here.

Intuitively, in order to build a query tableau q from a query tree $\mathcal{T}$, one associates a row to each leaf node and then rewrites the condition using the function h_{Att}. Then, the TED Why-Not answer for the query tree $\mathcal{T}$ is defined as the TED Why-Not answer for q. Thus, of course, two query trees sharing the same tableau representation have the same TED Why-Not answer.

Now, let us explore the other direction in order to characterize the set of query trees equivalent w.r.t. the TED Why-Not answer. We start by associating a class of query trees to a query tableau q.

Definition 3.7 (Query trees w.r.t. q**)** *Let q=$(\_, T_q, C_q)$ be a query tableau and assume that $|\mathcal{S}_q|$=n. The set $opSet$ of tree operators associated with q is the set of selections $\{\sigma_{h^{-1}_{Att}(c)} | c \in C_q\}$.*
A query tree $\mathcal{T}$ is associated with q iff (i) it has exactly $n-1$ cross product nodes, (ii) it has exactly one node for each selection in $opSet$, (iii) it has exactly one leaf node for each relation R in $\mathcal{S}_q$, and finally, (iv) it is equivalent to q.

Intuitively, the difference between two trees $\mathcal{T}_1$ and $\mathcal{T}_2$ associated to the same query q is the order of the operators in the trees.

Theorem 3.1 *Given a query q, TED Why-Not answer is unique up to isomorphism for all possible query trees associated to q.*

The above theorem states that two equivalent query trees obtained by some reordering of their operators lead to the same Why-Not answer. Clearly, this behavior is more robust than the behavior of NedExplain, where the Why-Not answer may differ for every equivalent query tree.

The question about other (equivalent) query trees sharing the same Why-Not answer property remains open although we have investigated several directions. We have considered minimization of the tableau leading to equivalent query trees. We also have considered saturating the query conditions, once again leading to equivalent query trees. However, these query trees do not produce the same Why-Not answer as counterexamples show.

Subsumption of NedExplain result by TED Why-Not answer. The next result shows that the TED Why-Not answer subsumes the NedExplain Why-Not answer.

Theorem 3.2 *Let q be query tableau over the shema $\mathcal{S}_q$ and $\mathcal{I}$ be an instance over $\mathcal{S}_q$. Let t_c be a simple Why-Not question. Assume that $\mathcal{T}$ is a query tree representation of q. Let*

$$NED = \{\mathcal{T}' | \mathcal{T}' \textit{ is a subtree in } \mathcal{T}\}$$

be the NED Why-Not answer w.r.t. $\mathcal{T}$, $\mathcal{I}$ and t_c, and let

$$TED = \{x | x \textit{ is a combination in } TWNA(q, t_c, \mathcal{I})\}$$

be the TED Why-Not answer w.r.t. q, $\mathcal{I}$ and t_c. Then, we have that

$$\forall \mathcal{T}' \in NED\ \exists x \in TED\ s.t.\ c \in x$$

where c is the condition of the rooting operator of the subtree $\mathcal{T}'$.

4 Naive Ted Algorithm

This section summarizes the *Ted* algorithm that naively computes the Why-Not answer polynomial by implementing in a straightforward manner what we discussed in Sec. 3. Further details are available in [4].

Briefly, *Ted* firstly computes all the cc-tuples w.r.t. the Why-Not question and then for each cc-tuple, it identifies the picky query conditions, constructing along the way the Why-Not answer polynomial. It is easy to see that such a straightforward implementation is computationally prohibitive, as it implies computing and enumerating the set of cc-tuples $CCT(t_c, \mathcal{I})$ (Def. 3.3).

In principle, $CCT(t_c, \mathcal{I})$ can be computed by executing an SQL statement with the conditions of t_c over $\mathcal{I}$. This is however a costly query, as it requires in most cases performing cross products of subsets of the input instance relations. To counter this problem, we introduce the *valid partitioning of* T_{t_c} that first computes sets of *partial cc-tuples* efficiently, which then need to be combined.

Definition 4.1 *(Valid Partitioning of T_{t_c}). The partitioning of T_q into k partitions $Part_1, \ldots, Part_k$, denoted $Partitioning = \{Part_i, \ldots, Part_k\}$, is* valid *for T_{t_c} if each $Part_i$ is minimal w.r.t. the following property:*
if $R{\in}Part_i$ and $R'{\in}\mathcal{S}_q$ s.t. $\exists c{\in}C_q$ with $h_{Att}(var(c)){\cap}Att(R){\neq}\emptyset$ and $h_{Att}(var(c)){\cap}Att(R){\neq}\emptyset$ then $R'{\in}Part_i$.
A tuple $\tau{\in}CCT(t_{c|Part_i}, \mathcal{I}_{|Part_i})$ is called a partial *cc-tuple.*

Example 4.1 *In our example, the valid partitioning (see Tab. 3) is $Part_1{=}\{R, S\}$ (because of the condition $x_2{=}x_4$, where x_2, resp. x_4 refers to R.B, resp. S.D) and $Part_2{=}\{T\}$. Examples of partial cc-tuples include $Id_1Id_5{\in}CCT(t_{c|Part_1}, \mathcal{I})$ and $Id_8{\in}CCT(T_{t_c|Part_2}, \mathcal{I})$.*

It is easy to prove that the valid partitioning of T_{t_c} is unique and the following lemma states how to compute $CCT(t_c, \mathcal{I})$ based on partial cc-tuples.

Lemma 4.1 *Let $\mathcal{P}{=}\{Part_1, \ldots, Part_k\}$ be the valid partitioning of T_{t_c} and $\mathcal{I}$ database instance over $\mathcal{S}_q$. Then,*
$CCT(t_c, \mathcal{I}){=}\bigtimes_{Part_i \in \mathcal{P}} CCT(t_{c|Part_i}, \mathcal{I}_{|Part_i})$.

Although this partitioning helps to reduce the computation and materialization of cross products, *Ted*'s worst case time complexity remains $O(n^{|\mathcal{S}_q|})$, $n{=}max(\{|\mathcal{I}_R|\ |R{\in}\mathcal{S}_Q)$. So, as validated also by experiments in Sec. 6, *Ted* is not of practical interest.

5 Efficient Ted++ Algorithm

The main feature of *Ted*++ is to completely avoid cross product materialization, thus significantly reducing both space and time consumption. To achieve this, *Ted*++ performs two main paradigm shifts. First, instead of tracing *picked* cc-tuples, it focuses on tracing *passing* cc-tuples. Second, *Ted*++ starts with a "polynomial template" that includes *all possible* condition combinations (addends) with variable coefficients to then incrementally and mathematically compute the coefficients for these addends.

Alg. 1 presents the main steps of *Ted*++. *Ted*++'s input includes the query $q{=}(T_q, C_q)$, the Why-Not question t_c and the input database instance $\mathcal{I}$. The following subsections discuss the individual steps of the algorithm in more detail.

5.1 Preprocessing

The first step of *Ted*++ (Alg. 1, line 1) is the computation of the "polynomial template" mentioned above, which is simply obtained by computing the power set of the query condition set C_q, from which the empty set is discarded.

Example 5.1 *For our example (Fig. 3), the search space is $\{op_2, op_3, \ldots, op_2op_3, \ldots, op_2op_3op_4op_5op_6\}$; its size is $2^5 - 1$.*

Algorithm 1: *Ted++*

```
  Input: query q, instance I, Why-Not question t_c
  Output: WNAPoly, the Why-Not answer polynomial
1 Initialization of CombSet, and T_{t_c} %preprocessing
2 Partition ←findValidPartitioning(T_{t_c}); (Def. 4.1)
3 for Part in Partition do
4     DB← Materialize V_Part % based on T_{t_c|Part};
5 CombSet ←PickyCombinations(CombSet, DB);
6 WNAPoly ←ExactAnswer(CombSet);   %postprocessing
7 return WNAPoly;
```

All along the algorithm, *Ted++* maintains a data structure called $CombSet$. For each condition combination x it registers a tuple $comb_x$=($opSet, partS, V, \#Pick$) where $opSet$ contains the conditions in x, $partS$ is a set of partitions (defined later on), V is a view definition meant to store the passing partial cc-tuples w.r.t. x. Finally, $\#Pick$ is the number of picked cc-tuples w.r.t. x and all its super combinations (i.e., combinations containing x). To simplify the discussion, we refer to $\#Pick_x$ as the number of picked cc-tuples w.r.t. x. The 'exact number' of picked cc-tuples w.r.t. x is computed in a postprocessing step (Alg. 1, line 6).

As in *Ted*, the second preprocessing step builds the conditional tableau T_{t_c} as describe in Sec. 3.3.

5.2 Partial CC-Tuples Computation

To compute the partial cc-tuples, the tableau T_{t_c} is first partitioned according to Def. 4.1 (Alg. 1, line 2). Each partition $Part$ is associated with a view V_{Part}, called *partition view*. V_{Part} is defined as the query corresponding to the tableau $T_{t_c|Part}$ (see example below) and is materialized in the database (line 4).

Example 5.2 *We are given $Part_1$=$\{R, S\}$ and $Part_2$=$\{T\}$ (see Ex. 4.1). The view definition V_{Part_1} relies on the partial tableau $T_{t_c|Part_1}$ except for the outmost projection. The projected attributes are those constrained by the query q, plus the relation Ids in $Part_1$, that is $\{R.A, R.B, S.D, S.E, R\_Id, S\_Id\}$. Similarly for $Part_2$, the set of projected attributes is $\{T.C, T.D, T.T\_Id\}$. This results in the query definitions given below and the materializations shown at the bottom of Fig. 4.*

V_{Part_1}	V_{Part_2}

`SELECT R_Id, S_Id,`
`R.A,R.B,S.D,S.E`
`FROM R, S`
`WHERE R.B < S.D`
	`SELECT T_Id,T.B,T.C,T.D`
	`FROM T`
	`WHERE T.C ≤ 9`

5.3 Picky Condition Combinations

The next step (line 5) of Alg. 1 identifies *picky* condition combinations. The pseudo-code of the function $PickyCombinations$ is given in Alg. 2.

First, the set of condition combinations $CombSet$ is traversed in ascending order of condition combination size (i.e., from size 1 to $|C_q|$, see Alg. 2, line 1). For each combination x we compute the number of picked cc-tuples $\#Pick_x$, using in each iteration results obtained in previous ones. To calculate $\#Pick_x$ we rely on the calculation of the number of picked partial cc-tuples $|PPick_x|$. To do that, we rely on the set of partitions $partS_x$ in $comb_x$.

Let x be an atomic combination, i.e., one condition *op*. When *op* is simple, it refers exactly to one relation R, which belongs to one partition $Part$ hence $partS_{op}$=$\{Part\}$. When *op* is complex, it refers to two relations R and S and either both relations belong to the same partition $Part$ and $partS_{op}$=$\{Part\}$ or they belong to different partitions $Part_1$ and $Part_2$ and $partS_{op}$=$\{Part_1, Part_2\}$. Recall that, each partition $Part$ is associated with a partition view V_{Part} that stores the partial cc-tuples over $Part$. We

Algorithm 2: PickyCombinations

Input: $CombSet, DB$
Output: $CombSet$
1 **for** *k=1 to* $|C_q|$ **do**
2   **for** $comb_x \in CombSet$ *s.t.* $|opSet_x|=k$ **do**
3   Compute $partS_x$;
4   **if** *k=1* **then**
5       $DB \leftarrow$Materialize V_x; (Def. 5.1)
6   **else**
7   **if** V_x *needs to be materialized* **then**
8   ($comb_{x_1}$,$comb_{x_2}$) $\leftarrow$ SelectSubCombinations(V_x);
9   **if** *Target schemas of* V_{x_1} *and* V_{x_2} *share common attributes Att* **then**
10           $V_x \leftarrow V_{x_1} \bowtie_{Att} V_{x_2}$;
11           $DB \leftarrow$Materialize V_x;
12   **else**
13         $|V_x| \leftarrow$multiply sizes of sub-combination views in x;
14     $|PPick_x| \leftarrow$Apply Equ. (G);
15     $\#Pick_x \leftarrow$Apply Equ. (A);
16 **return** $CombSet$;

associate with $PartS_{op}$ the set of partition views $\mathcal{V}=\{V_{Part}|Part \in PartS_{op}\}$. We now generalize to a non atomic condition combination x where $opSet_x \subseteq C_q$ and define $partS_x = \cup_{op \in opSet_x} partS_{op}$.

Example 5.3 *Consider the two atomic combinations* op_2 $(x_1>3)$ *and* op_3 $(x_2=x_6)$. *Looking at Tab. 3, we see that* op_2 *refers only to* $Part_1$, *whereas the variables of* op_3 *span over* $Part_1$ *and* $Part_2$. *Hence,* $partS_{op_2}=\{Part_1\}$ *and* $partS_{op_3}=\{Part_1, Part_2\}$. *Considering the condition combination* x *where* $opSet_x=\{op_2 op_3\}$, *we obtain* $Part_x=\{Part_1, Part_2\}$. *Fig. 4 associates to all atomic combinations their respective sets* $Part_{op}$ *using edges between op and partition views.*

Using $partS_x$, the number of picked cc-tuples w.r.t. combination x, i.e., $\#Pick_x$ is computed by Equ. (A):

$$\#Pick_x = |PPick_x| \times \prod_{Part \in \overline{partS_x}} |V_{Part}|, \tag{A}$$

where $\overline{partS_x}=Partitioning \setminus partS_x$. Note that when $\overline{partS_x}$ is empty, we abusively consider that $\prod_{\emptyset}=1$. Intuitively, the above formula extends the *partial* cc-tuples to "full" cc-tuples over all partition schemas.

The presentation now focuses on calculating $\#Pick_x$, by firstly calculating $|PPick_x|$. Two cases arise depending on the size of the condition combination.

Atomic condition combinations. We start with considering condition combinations x s.t. $|opSet_x|=1$ (Algorithm 2, Line 5). To find the number $PPick_x$ of picked partial cc-tuples w.r.t. x we compute and materialize the set of passing partial cc-tuples through the query provided below:

Definition 5.1 *(Condition View.) Let op be a condition,* $partS$ *its associated set of partitions and* $\mathcal{V}$ *its associated set of partition views. Then, the condition view* V_{op} *for op is specified by:*

$$V_{op} = \begin{cases} \pi_{\{R_{id}|R \in Part\}}(\sigma_{op}[V_{Part}]) \textit{ if } partS=\{Part\} \\ \pi_{\{R_{id}|R \in Part_1 \cup Part_2\}}([V_{Part_1}] \bowtie_{op} [V_{Part_2}]) \\ \qquad \textit{if } partS=\{Part_1, Part_2\} \end{cases}$$

Example 5.4 *Given* $PartS_{op_3}=\{Part_1, Part_2\}$, V_{op_3} *is*

$$\pi_{Part_1.R\_Id, Part_1.S\_Id, Part_2.T\_Id}([V_{Part_1}] \bowtie_{R.B=T.B} [V_{Part_2}])$$

Fig. 4 shows the materialization of all operator views. Note that the materialization of view V_2 is empty, hence, no associated materialization is presented.

We can now compute the number of picked partial cc-tuples for an operator *op* by:

$$|PPick_{op}| = \prod_{Part \in partS_{op}} |V_{Part}| - |V_{op}| \tag{B}$$

Example 5.5 *For* op_3*, we have* $|V_{op_3}|$=4*. So,* $|PPick_{op_3}|$=$|V_{Part1}|$×$|V_{Part2}|$−$|V_3|$=3×4−4=8*. Since all partitions of* $Partitioning$ *are in* $partS_{op_3}$*, applying Equ. (A) results in* $\#Pick_{op_3}$=$PPick_{op_3}$=8*. Opposed to that, for* op_4*,* $|PPick_{op_4}|$=$|V_{Part_2}|$−V_{op_4}=4−2=2*, so* $\#Pick_{op_4}$=$2 * 3 = 6$*. The results of Equ. (A) and (B) are shown in Fig. 4 for all remaining atomic combinations.*

Non atomic condition combinations. After processing all atomic conditions, we proceed with non-atomic ones (Alg. 2, lines 6-13).

Let $opSet_x$=$\{op_i|i$=$1 \ldots N\}$ be the set of conditions of the combination x. Intuitively, to find the picked partial cc-tuples w.r.t. x, we need to find the picked partial cc-tuples common to op_1 and … and op_N. These common cc-tuples are in the intersection of the sets of picked partial cc-tuples of $op_1 \ldots op_N$, stored in the materialization of $V_{op_1} \ldots V_{op_N}$. As we will see, in order to compute the intersection (or simply get its cardinality), we may have to use, in addition to the condition views V_{op_i}, all views associated with the sub-combinations of x, including x itself.

To describe this most complicated step of the algorithm, we start by developing a simplified case. Assume that all condition views have the same target schema Att_x=$\{R\_Id|R$∈P and P∈$partS_x\}$.

The set of picked partial cc-tuples w.r.t. x is computed as the intersection of the complements of the condition views storing the passing partial cc-tuples:

$$PPick_x = \overline{V_{op_1}} \cap \cdots \cap \overline{V_{op_N}} = \overline{V_{op_1} \cup \cdots \cup V_{op_N}} \tag{C}$$

As our design decision was to only materialize passing partial cc-tuples of views V_i, we rewrite $PPick_x$ as:

$$PPick_x = \pi_{Att_x}[\bigtimes_{Part \in partS_x} V_{Part}] \setminus \bigcup_{op \in opSet_x} V_{op} \tag{D}$$

Given the assumption that all condition views have the same schema, applying the set operators (difference, union) is well defined, and so is the query $PPick_x$. However, in the general case, this assumption does not hold. Thus, to deal with the general case, the previous equations need to be rewritten by "extending" condition views V_{op} to views V_{op}^{ext} over a common schema with attributes Att_x (as defined above):

$$V_{op}^{ext} = \pi_{Att_x \setminus Att_{op}}[\bigtimes_{Part \in partS_x \setminus partS_{op}} V_{Part}] \times V_{op} \tag{E}$$

This extended view substitutes V_{op} in Equ. (D) in the general case and we thus obtain the following query to compute $PPick_x$:

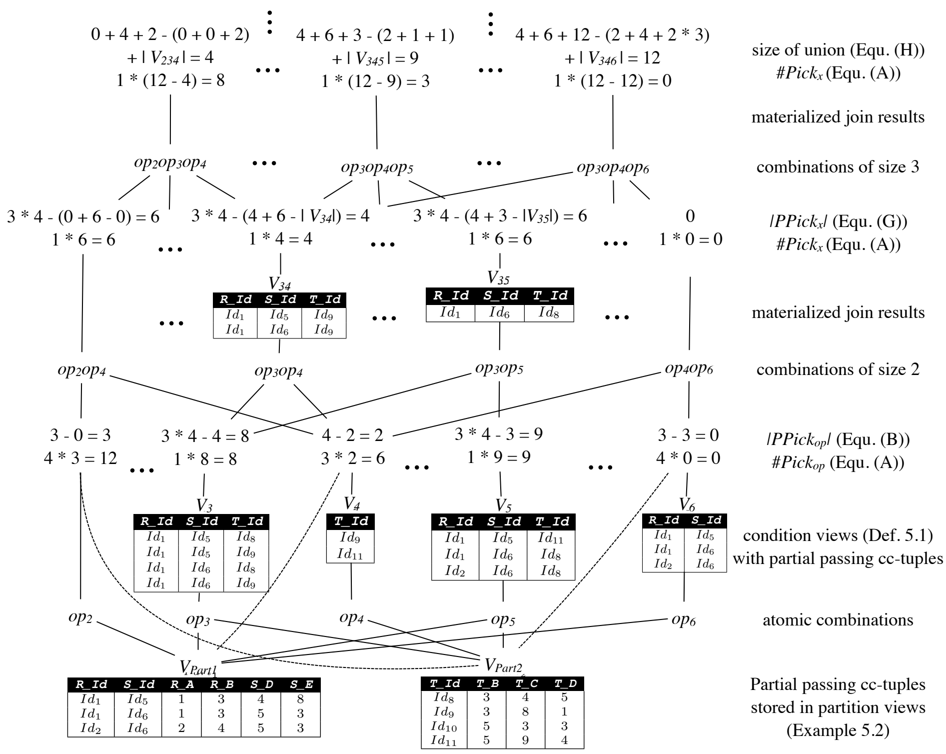
$$PPick_x = \pi_{Att_x}[\bigtimes_{Part \in partS_x} V_{Part}] \setminus \bigcup_{op \in opSet_x} V_{op}^{ext} \tag{F}$$

As already said, the main feature of *Ted++* is to avoid computing cross products, so clearly, we do not want to compute the cross product introduced in Equ. (E) and (F). Fortunately, remember that we are not interested in the set of picked cc-tuples itself, we only require its cardinality, which we can compute as follows.

$$| PPick_x |= \prod_{Part \in partS_x} |V_{Part}| - | \bigcup_{op \in opSet_x} V_{op}^{ext} | \tag{G}$$

Equ. (G) generalizes Equ. (B) introduced for the atomic combinations. To calculate the size of the union, we apply the *Principle of Inclusion and Exclusion for counting* [12]:

$$| \bigcup_{i=1}^{N} V_{op_i}^{ext} |= \sum_{\emptyset \neq \Gamma \subseteq [N]} (-1)^{|\Gamma|+1} | \bigcap_{\gamma \in \Gamma} V_{op_\gamma}^{ext} | \tag{H}$$



V_{34}:

R_Id	S_Id	T_Id
Id_1	Id_5	Id_9
Id_1	Id_6	Id_9

V_{35}:

R_Id	S_Id	T_Id
Id_1	Id_6	Id_8

V_3:

R_Id	S_Id	T_Id
Id_1	Id_5	Id_8
Id_1	Id_5	Id_9
Id_1	Id_6	Id_8
Id_1	Id_6	Id_9

V_4:

T_Id
Id_9
Id_{11}

V_5:

R_Id	S_Id	T_Id
Id_1	Id_5	Id_{11}
Id_1	Id_6	Id_8
Id_2	Id_6	Id_8

V_6:

R_Id	S_Id
Id_1	Id_5
Id_1	Id_6
Id_2	Id_6

V_{Part1}:

R_Id	S_Id	R_A	R_B	S_D	S_E
Id_1	Id_5	1	3	4	8
Id_1	Id_6	1	3	5	3
Id_2	Id_6	2	4	5	3

V_{Part2}:

T_Id	T_B	T_C	T_D
Id_8	3	4	5
Id_9	3	8	1
Id_{10}	5	3	3
Id_{11}	5	9	4

Figure 4: Running example illustrating the different steps of *Ted++* defined in Alg. 1 and Alg. 2

Example 5.6 *To illustrate the concepts introduced above, please follow on Fig. 4 the following discussion.*

For the combination op_3op_4, Equ. (H) gives: $|V_3^{ext} \cup V_4^{ext}|=|V_3^{ext}| + |V_4^{ext}| - |V_3^{ext} \cap V_4^{ext}|$[2]. The schema of $Part_{34}=\{Part_1, Part_2\}$ is $Att_{34}=\{R\_Id, S\_Id, T\_Id\}$. The view V_3 has already a matching schema, thus $|V_3^{ext}|=|V_3|=4$. For V_4, $Att_4=\{T\_Id\}$, we thus apply Eqn. (E) and obtain $|V_4^{ext}|=|V_{Part_1}|\times |V_4|=3 \times 2=6$. Still, $|V_{34}|=|V_3^{ext} \cap V_4^{ext}|$ remains to be calculated. Intuitively, because V_3 and V_4 target schemas share attribute T_Id, $V_{34}=V_3 \bowtie_{T\_Id} V_4$. The view V_{34} is materialized and contains 2 tuples (as shown in Fig. 4). So, finally, for Equ. (H) we obtain $|V_3^{ext} \cup V_4^{ext}|=4+6-2=8$, yielding for Equ. (G) $|PPick_{34}|=12-8=4$, and eventually $\#Pick_{34} = 4$ (Equ. (A)).

We now focus on combination op_4op_6. The schemas of V_4 and V_6 are disjoint and intuitively $V_{46}=V_4 \times V_6$. Here, V_{46} is not materialized, we simply calculate $|V_{46}|=|V_4|\times|V_6|=6$. Then,

[2]For brevity, we use subscript i instead of op_i in the following.

$|PPick_{46}|$=12−(12+6−6)=0, *which is expected as* op_6 *has no picked cc-tuples (see Ex. 5.5), so neither do any of its super-combinations.*

Finally, consider the combination $op_3op_4op_6$ *of size 3. Equ. (H) is then:* $|V_3^{ext} \cup V_4^{ext} \cup V_6^{ext}| = |V_3^{ext}| + |V_4^{ext}| + |V_6^{ext}| - |V_3^{ext} \cap V_4^{ext}| - |V_3^{ext} \cap V_6^{ext}| - |V_4^{ext} \cap V_6^{ext}| + |V_3^{ext} \cap V_4^{ext} \cap V_6^{ext}|$. *All terms of the right side of the equation are available from previous iterations, except for* $|V_3^{ext} \cap V_4^{ext} \cap V_6^{ext}|$. *As before, we check the common attributes of the atomic condition views and obtain* $V_{346}=V_6 \bowtie_{R\_Id,S\_Id} V_3 \bowtie_{T\_Id} V_4$. *The view is materialized and displayed in Fig. 4. We obtain* $|V_3^{ext} \cup V_4^{ext} \cup V_6^{ext}| = 4+6+12-(2+4+6)+2 = 12$ *and, as expected,* $|PPick_{346}|=0$. *This calculation can be further simplified to* $|V_3^{ext} \cup V_4^{ext} \cup V_6^{ext}| = |V_3^{ext} \cup V_4^{ext}| + |V_6^{ext}| - |V_3^{ext} \cap V_6^{ext}| - |V_4^{ext} \cap V_6^{ext}| + |V_3^{ext} \cap V_4^{ext} \cap V_6^{ext}|$.

Ex. 5.6 demonstrates that for a combination x, $|\bigcup_{i=1}^{N} V_{op_i}^{ext}|$ can be computed incrementally from $|\bigcup_{i=1}^{N-1} V_{op_i}^{ext}|$. Formally, for $N>1$

$$\begin{aligned} |\bigcup_{i=1}^{N} V_{op_i}^{ext}| = {} & |\bigcup_{i=1}^{N-1} V_{op_i}^{ext}| + |V_{op_N}^{ext}| \\ & + \sum_{\emptyset \neq \Gamma \subseteq [N-1]} (-1)^{|\Gamma|} \, |\bigcap_{\gamma \in \Gamma} V_{op_\gamma}^{ext} \cap V_{op_N}^{ext}| \end{aligned} \tag{I}$$

Using this final equation, we can now generally compute $\#Pick_x$ by first applying (G) and then Equ. (A).

View Materialization: when and how. Ex. 5.6 suggests that the view V_x may or may not be materialized. To decide on materialization (Alg. 2, line 7), we partition the set $\mathcal{V}_x$ of the views associated with the conditions in $opSet_x$. 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 $\mathcal{V}_x$ through $\sim^*$.

When this partitioning is a singleton, V_x needs to be materialized. The materialization of V_x 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 $op_3op_4op_5$, V_{345} can either be computed through $V_{34} \bowtie V_5$ or $V_{35} \bowtie V_4$ or $V_{45} \bowtie V_3$ $V_3 \bowtie V_4 \bowtie V_5$... because all these views are known from previous iterations. The choice of the query used to materialize V_x is done based on a cost function. This function gives priority to materializing V_x by means of one join, which is always possible: because V_x 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 $V_{34} \bowtie V_5$ or $V_{35} \bowtie V_4$ or $V_{45} \bowtie V_3$ is considered. In order to choose among the one-join queries computing V_x, we favor a one-join query $V_i \bowtie V_j$ minimal w.r.t. $|V_i|+|V_j|$. For the example, and considering also Fig. 4 we find that $|V_3|+|V_{45}|=$ $|V_5|+|V_{34}|=5$ and $|V_4|+|V_{35}|=3$. So, the query used for the materialization is $V_4 \bowtie V_{35}$ (its result being empty in our example).

If the partitioning is not a singleton, V_x is not materialized (line 13). For example, the partitioning for op_4op_6 is not a singleton and so the size $|V_{46}|=|V_4| \times |V_6|=6$.

Finally, we can avoid materialization even if the partitioning is a singleton, when for some sub-combination y of x it was found that $\#Pick_y=0$. In that case, we know a priori that $\#Pick_y=0$ (e.g., in Ex. 5.6, $\#Pick_6=0$ implies $\#Pick_{36}=0$, $\#Pick_{346}=0$ etc.).

5.4 Postprocessing

The result of Alg. 2 returns for each picky condition combinations y an associated coefficient $\#Pick_y$. However, recall that the calculation of this coefficient so far counts any cc-tuple picked by a combination

y to be also picked by any of its sub-combinations (see Ex. 5.7). Thus, the last step of *Ted++* is to compute, for each combination, the exact coefficient (see Alg. 1, line 6).

The exact coefficient for a combination x is obtained by subtracting the coefficients of its super-combinations from $\#Pick_x$:

$$coef_x{=}\#Pick_x{-}(\sum_{opSet_x \subseteq opSet_y} coef_y) \tag{J}$$

Example 5.7 *Consider known $coef_{2345}$=2 and $coef_{234}$=2. We have found in Ex. 5.6 that $\#Pick_{34}$=4. With Equ. (J), $coef_{34}$=4−2−2=0. In the same way $coef_3$=4−0−2−2=0. The algorithm leads to the expected Why-Not answer polynomial already provided in Ex. 3.7.*

5.5 Theoretical Discussion of Ted++

Theorem 5.1 states that *Ted++* (Alg. 1) is sound and complete w.r.t. Def. 3.6.

Theorem 5.1 *Given a query q, a Why-Not question t_c and an input instance $\mathcal{I}$,* Ted++ *computes exactly $TWNA(q, t_c, \mathcal{I})$.*

Complexity analysis. In the pseudo-code for *Ted++* provided in Alg. 1, we can see that *Ted++* divides into the phases of (i) partitioning T_{t_c}, (ii) materializing a view for each partition, (iii) computing picky combinations, and (iv) computing the exact coefficients. When computing picky combinations, according to Alg. 2, *Ted++* iterates through $2^{|C_q|}$ condition combinations and for each, it decides upon view materialization (again through partitioning) before materializing it, or simply calculates $|V_x|$ before applying equations to compute $\#Pick$. Overall, we consider that all mathematical computations are negligible so, the worst case complexities of steps (i) through (iv) are $O(|S_q|{+}|t_c.cond|){+}O(|S_q|) + O(2^{|C_q|}(|S_q| + |C_q|)){+}O(2^{|C_q|})$. For large enough queries, we can assume that $|S_q|{+}|C_q|{<<}2^{|C_q|}$, in which case the complexity simplifies to $O(2^{|C_q|})$.

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 $n{=}max(\{|\mathcal{I}_R||R{\in}S_q\})$. 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 $O(n^{|S_q|})$. This yields a combined complexity of $O(2^{|C_q|}n^{|S_q|})$. However, *Ted++* in the general case (more than one induced partitions), has a tighter upper bound: $O(n^{k_{x1}} + n^{k_{x2}} + \cdots + n^{k_{xN}})$, where $k_x{=}|\{Part|Part{\in}partS_x\}|$, for all combinations x and $N = 2^{|C_q|}$. It is easy to see that $n^{k_{x1}}{+}n^{k_{x2}}{+}\ldots{+}n^{k_{xN}} < 2^{|C_q|}n^{|S_q|}$, when there is more than one partition.

6 Experimental Evaluation

This section presents an experimental evaluation of *Ted++*. In Sec. 6.1, we compare *Ted++* with the existing algorithms returning query-based explanations, i.e., with NedExplain [5] and Why-Not [7]. The comparison shows that the runtime of *Ted++* is competitive with the runtime of these algorithms, while computing a more informative answer. Sec. 6.2 studies the runtime of *Ted++* with respect to various parameters that we vary in a controlled manner. Overall, *Ted++* scales well with respect to the studied parameters, demonstrating *Ted++*'s practicality.

We have implemented *Ted*, *Ted++*, NedExplain, and Why-Not in Java. The original Why-Not implementation, as well as ours, relies on the lineage tracing provided by Trio (http://infolab.stanford.edu/trio/). We ran 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 PostegreSQL 9.3 as database system.

Table 5: Queries for the scenarios in Tab. 6

Query	Expression
Q1	$C \bowtie_{sector} W \bowtie_{witnessName} S \bowtie_{hair,clothes} P$
Q2	$\sigma_{C.sector>99}[C] \bowtie_{sector} W \bowtie_{witnessName} S \bowtie_{hair,clothes} P$
Q3	$W \bowtie_{sector2} C2 \bowtie_{sector1} \sigma_{C.type=Aiding}[C]$
Q4	$P2 \bowtie_{!name,hair} \sigma_{P1.name<B}[P1]$
Q5	$L \bowtie_{movieId} \sigma_{M.year>2009}[M] \bowtie_{name} \sigma_{R.rating \geq 8}[R]$
Q6	$\sigma_{AA.party=Republican}[AA] \bowtie_{id} \sigma_{Co.Byear>1970}[Co]$
Q7	$E \bowtie_{eId} \sigma_{ES.sub=Sen.\ Com.}[ES] \bowtie_{id} \sigma_{SPO.party=Rep.}[SPO]$
Q_{s3}	$\sigma_{type=Aiding}[Q2]$
Q_{s4}	$\sigma_{witnessname>S}[Q_{s3}]$
Q_j	$C \bowtie_{sector} \sigma_{name>S}[W]$
Q_{j2}	$Q_j \bowtie_{witnessname} S$
Q_{j3}	$Q_{j2} \bowtie_{clothes} P$
Q_{j4}	$Q_{j3} \bowtie_{hair} P$
Q_c	$L1 \bowtie_{lid} L2 \bowtie_{M2.mid=L2.mid} M2 \bowtie_{year,!mid} \sigma_{year=1980}[M1]$
Q_{tpch}	$C \bowtie_{ckey} \sigma_{odate<1998-07-21}[O] \bowtie_{okey} \sigma_{sdate>1998-07-21}[L]$

6.1 Comparative Evaluation

We begin the evaluation of *Ted*++ with the comparative evaluation to algorithms Why-Not and NedExplain. This evaluation considers both efficiency (runtime) and effectiveness (Why-Not answer quality) of the different algorithms. When considering efficiency, we also include *Ted* in the comparison (*Ted* producing the same Why-Not answer as *Ted*++).

Experimental Setup. For the experiments in this section, we have used data from three databases named *crime*, *imdb*, and *gov*. The *crime* database corresponds to the sample crime database of Trio and was previously used to evaluate Why-Not and NedExplain. The data describes crimes and involved persons (suspects and witnesses). The *imdb* database is built on real-world movie data extracted from IMDB (http://www.imdb.com). Finally, the *gov* database contains information about US congressmen and financial activities[3]. The table sizes in the datasets range from 89 to 9341 records.

For each dataset, we have created a series of scenarios (crime1-gov5 in Tab. 6). Each scenario consists of a query further defined in Tab. 5 (Q1-Q7) and a simple Why-Not question, as all algorithms but *Ted*++ 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).

6.1.1 Why-Not Answer Evaluation

In our discussion of related work (summarized in Tab. 1), we have seen that Why-Not and NedExplain return query operators, whereas *Ted*++ returns a polynomial where each addend includes a condition combination. For comparison purposes, we trivially map *Ted*++'s Why-Not answer to a set of operator sets, e.g., $3op_3 * op_4 + 2op_3 * op_6$ maps to $\{\{op_3, op_4\}, \{op3, op_6\}\}$. For conciseness, we abbreviate operator sets, e.g., to op_{34}, op_{36}.

Tab. 7 summarizes the Why-Not answers of the three algorithms. The following discussion focuses on comparing *Ted*++to Why-Not and NedExplain, as a detailed comparison between these two has already been provided in [5].

For all the tested scenarios, we observe that all operators identified by NedExplain or Why-Not also exist in the answer of *Ted*++, either as atomic combinations or as part of a combination. For instance, in crime5, Why-Not returns op_5 ($\sigma_{sector>99}[C]$) whereas NedExplain returns op_1 ($C \bowtie_{sector} W$). These Why-Not answers are subsumed by the Why-Not answer *Ted*++ returns, i.e., the polynomial includes both the atomic combination op_5 as well as a combinations including op_1, e.g., op_{15}. If a picky operator matches an atomic condition, e.g., op_5, this means that fixing this operator will be sufficient to produce

[3]Collected at http://bioguide.congress.gov, http://usaspending.gov, and http://earmarks.omb.gov

Table 6: Scenarios

Scenario	Query	Why-Not question
crime1	Q1	(P.Name:Hank,C.Type:Car theft)
crime2	Q1	(P.Name:Roger,C.Type:Car theft)
crime3	Q2	(P.Name:Roger,C.Type:Car theft)
crime4	Q2	(P.Name:Hank,C.Type:Car theft)
crime5	Q2	(P.Name:Hank)
crime6	Q3	(C2.Type:kidnapping)
crime7	Q3	(W.Name:Susan,C2.Type:kidnapping)
crime8	Q4	(P2.Name:Audrey)
imdb1	Q5	(name:Avatar)
imdb2	Q5	(name:Christmas Story,L.locationId:USANew York)
gov1	Q6	(Co.firstname:Christopher)
gov2	Q6	(Co.firstname:Christopher,Co.lastname:MURPHY)
gov3	Q6	(Co.firstname:Christopher,Co.lastname:GIBSON)
gov4	Q7	(sponsorId:467)
gov5	Q7	((SPO.sponsorln:Lugar,E.camount:x),x>=1000)
crime$_s$ − crime$_{s4}$	Q1,Q2, Q$_{s3}$,Q$_{s4}$	(P.Name:Hank,C.Type:Car theft)
crime$_j$ −crime$_{j4}$	Q$_j$ −Q$_{j4}$,	(W.name=Jane, C.type=Car theft)
imdb$_c$	Q$_{c4}$	(L2.locationid=L1.locationid, M1.mid=L2.mid, L1.year>L2.year,M1.name=Duck Soup)
imdb$_{c2}$	Q$_{c4}$	(L2.locationid=L1.locationid, M1.mid=L2.mid, L1.year>L2.year)
crime$_{5c2}$	Q2	(P.Name:Hank, C.type=Car theft)
crime$_{5c3}$	Q2	(P.Name:Hank, C.type=Car theft, S.witness=Aphrodite)
crime$_{5c4}$	Q2	(P.Name:Hank ,C.type=Car theft, S.witness=Aphrodite, W.sector =34)
crime$_{5c5}$	Q2	(P.Name:Hank ,C.type=Car theft, S.witness=Aphrodite, W.sector =34,S.hair=green)
imdb$_{cc}$	Q$_c$	(M.year>M2.year)
tpch$_s$	Q$_{tpch}$	(L.extprice>50000,O.odate<1996-01-01)
tpch$_c$	Q$_{tpch}$	(L.extprice>100000, O.odate=L.cdate, C.nkey=4)

the specified missing answer. On the contrary, if a picky operator only appears as part of a condition combination, e.g., op_1, *Ted++* provides us with the full explanation that requires fixing that operator in combination with others (fixing it alone will not make the missing answer appear).

Two special cases where the subsumption does not directly hold are crime2 and crime3. The reported answer of Why-Not and NedExplain contains the operator op'_{34}, which stands for $S \bowtie_{hair,clothes} P$. While NedExplain and Why-Not consider this as one operator, *Ted++*, as a consequence of Def. 3.7, has two complex conditions for this, i.e., op_3 ($S.hair{=}P.hair$) and op_4 ($S.clothes{=}P.clothes$). In this case, op'_{34} maps either to op_3, op_4, or op_{34} without being more precise. Again, *Ted++* is more informative here as it clearly indicates which of these combinations are indeed culprit, for example op_3 ($S.hair{=}P.hair$) alone is picky for $crime2$ but not for crime3.

Considering gov2, recall that Why-Not and NedExplain rely on a query tree. For this scenario, the trees chosen by Why-Not and NedExplain actually differ. Why-Not identifies op_1, whereas NedExplain identifies op_3 as the picky operator. *Ted++* contains both operators as atomic picky combinations in its result, showing also experimentally its independence from the query tree representation.

Another interesting case is crime8. NedExplain indicates that op_2 ($S \bowtie_{hair} P$) is picky, but *Ted++* also computes the picky atomic combination op_3 ($\sigma_{name<`B'}[P]$). From a developer's perspective, selections are typically easier or more reasonable to change, so she would typically start fixing these. But here, NedExplain does not even give her the information that trying to fix the selection may be successful. Thus, it is not only a matter of whether NedExplain or Why-Not produce a correct answer, but also which correct answer. With *Ted++* the developer gets all necessary information to decide what fixes to test first.

In gov3, NedExplain and Why-Not both return op_2. However, let us now assume the developer is not willing to change this operator. So, remembering that the algorithms' answers may change when changing the query tree, she may start trying different options to possibly obtain a different Why-Not

Table 7: *Ted*++, Why-Not, NedExplain answers per scenario

Scenario	***Ted*++**	**Why-Not**	**NedExplain**
crime1	$op_{1234}, \ldots, op_{12}, op_3, op_2, op_1$		op_1
crime2	$op_{1234}, op_{34}, op_{13}, \ldots, op_3$	op'_{34}	op'_{34}, op_1
crime3	$op_{12345}, \ldots, op_{145}, op_{345}, op_{35}$	op'_{34},op_5	op_5, op'_{34}
crime4	$op_{12345}, \ldots, op_{25}, op_{15}$	op_5	op_1, op_5
crime5	$op_{12345}, \ldots, op_{15}, op_5$	op_5	op_1
crime6	$op_{123}, op_{31}, op_{23}, op_{12}, op_3, op_2, op_1$	op_3	op_2
crime7	$op_{123}, op_{13}, op_{12}, op_1$	op_3	op_2, op_1
crime8	$op_{23}, op_3, op_2, op_1$		op_2
imdb1	$op_{123}, op_{13}, op_{23}, op_3$	op_3	op_3,op_2
imdb2	op_{13}		op_1, op_3
gov1	$op_{123}, op_{13}, op_{23}, op_{12}, op_3, op_2, op_1$	op_3	op_2, op_3
gov2	op_{13}, op_3, op_1	op_1	op_3
gov3	op_{123}, op_{23}, op_2	op_2	op_2
gov4	op_{123}, op_{23}, op_2	op_3	op_3, op_2
gov5	$op_{124}, op_{14}, op_{24}, op_{12}, op_4, op_2, op_1$	op_1	op_1

answer. Looking at the answer of *Ted*++ would prevent her from spending any effort on this, as it shows that each condition combination includes op_2.

So far, our discussion focused on the value of providing all condition combinations, but valuable information is obtained also by the coefficients. For an example, for crime8, the complete Why-Not answer polynomial is $2384 * op_{23} + 20 * op_3 + 4 * op1 + 8 * op_2$. Assume that the developer has no preference on which condition to change, but she wants to minimize changes while maximizing chances of getting the missing answer in the query result. Looking at the polynomial, it is easy to see that minimal changes means changing one of op_1,op_2, and op_3 while the highest coefficient of these three atomic combinations indicates the maximized chances of getting the missing answer in the result. Thus, she would choose op_3. Clearly, the results of NedExplain or Why-Not do not provide sufficient information to make such an informed decision.

6.1.2 Runtime Evaluation

We now compare the runtime of *Ted*++ with other algorithms.

***Ted*++ vs. NedExplain and Why-Not.** For this comparative evaluation, we again consider scenarios crime1 through gov5 of Tab. 6 as their use of simple Why-not questions ensures that they are supported by all three algorithms. Fig. 5 summarizes the runtimes in logarithmic scale for each algorithm and each 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. We explain this behavior as follows.

Why-Not traces compatible tuples based on tuples' lineage stored in the Trio system. As already stated in [7] and [5], this design choice slows down Why-Not performance. Opposed to that, both NedExplain and *Ted*++compute the compatible data more efficiently by issuing a few simple select SQL statements

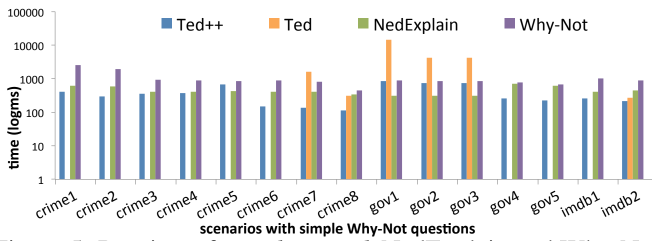


Figure 5: Runtimes for *Ted*++, *Ted*, NedExplain and Why-Not

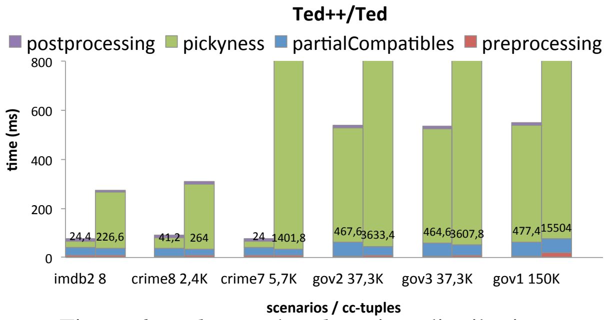


Figure 6: *Ted*++ and *Ted* runtime distribution

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 limiting to compatible data in the instance. 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 scenarios gov1–gov3. In these scenarios, all compatible tuples are picked by operators very close to the leaf level of the operator tree, so the bottom-up traversal of the tree can stop very early. *Ted*++ will always “check” all conditions so cannot benefit from such an early termination. However, this runtime improvement opportunity in NedExplain often comes at the price of reduced information conveyed by the Why-Not answer (e.g., a partial Why-Not answer in gov1).

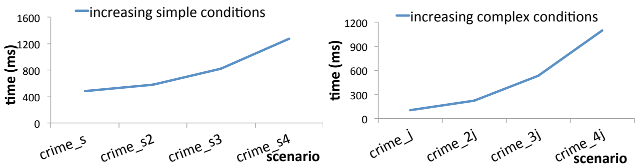
***Ted*++ vs. *Ted*.** Fig. 5 also reports runtimes for *Ted* on 6 out of 15 scenarios (all others did not run). To experimentally demonstrate where *Ted*’s problem lies, we compare the time distribution of different algorithm phases in *Ted* and *Ted*++ for these scenarios.

Fig. 6 divides the runtime into four common phases of the algorithms. Among these, the pickyness phase is the one that is inherently different in both algorithms. *Ted* iterates over the whole cc-tuple set and computes the picky condition combinations for each cc-tuple. *Ted*++ explores the search space, and calculates the number of picked cc-tuples per condition combination. 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 pickyness phase is provided as label on the respective bars).

As said before, *Ted*’s main issue w.r.t. efficiency is its strong dependence on the number of cc-tuples. This is experimentally observed in Fig. 6: with the growth of the set of cc-tuples in the scenarios, the time dedicated to pickyness also grows (the scenarios are reported in an ascending order). *Ted*++ depends on the number of cc-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 cc-tuples grows, *Ted*++’s pickyness phase remains roughly steady.

6.2 Ted++ Investigation

We now study *Ted*++’s behavior when varying the following parameters: (i) the type (simple or complex) of the input query q and the number of its conditions, (ii) the type of the Why-Not question (simple or complex) and the number and selectivity of conditions it entails, and (iii) the size of the input database $\mathcal{I}$. Note that (ii) and (iii) are tightly connected with the number of computed cc-tuples, which is one of the main parameters influencing the performance. In addition to the number of cc-tuples, (i) determines the pickyness phase performance depending also on the selectivity of the query conditions over the compatible data.



(a) simple conditions (b) complex conditions

Figure 7: *Ted*++ runtime w.r.t. number of conditions in q

Experimental Setup. For the parameter variations (i) and (ii), we again use the *crime*, *imdb*, and *gov* databases. To adjust the database instance size for case (iii), we use data produced by theTPC-H[4] benchmark data generator. 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 operator.

The queries used in this section are summarized in Tab. 5 (Q$_s$-Q$_{tpch}$) and the scenarios in Tab. 6 (crime$_s$-tpch$_c$).

Adjusting the query q**.** 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, Q$_{s3}$, Q$_{s4}$. The evolution of runtime when applying *Ted*++ on this series of queries is shown in Fig. 7 (a). Similarly, starting from query Q$_j$, we introduce step by step complex conditions, yielding Q$_j$-Q$_{j4}$. 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 line depends on the selectivity of the introduced operator; the more selective the operator the steeper the line becomes. This is easy to explain, as in the pickyness phase, the operator view contains more tuples (=passing partial cc-tuples) when the operator is more selective. This results in more computations in the super-combinations iterations.

Note that the curve line in Fig. 7 (a) starts at point much higher than in Fig. 7 (b). This is because the query Q1 (crime$_s$) initially includes four complex conditions, in contrast to Q$_j$ (crime10) that includes one complex and one simple condition.

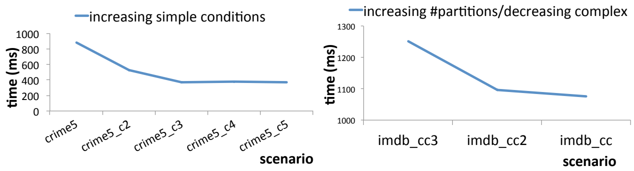
Adjusting the Why-Not question. Next, we vary the type and the number of conditions in the Why-Not question defined by t_c. Fig. 8 shows the cases when we start (a) with a simple t_c and progressively add more simple conditions and (b) start with a complex t_c and progressively add more complex conditions.

The scenarios considered for Fig. 8 (a) have as starting point the simple scenario crime5 (see Tab. 6). Then, keeping the same input instance and query, we add attibute-constant comparisons to t_c, a procedure resulting in fewer cc-tuples in each step. As expected, the more conditions (the less cc-tuples) the faster the Why-Not answer is returned, until we reach a certain fixpoint (here from crime5$_{c3}$ on). From this point on, the runtime is dominated by the time dedicated to communicate with the database that is constant over all scenarios.

As we introduce complex conditions to t_c, 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 cc-tuples constant* in our series of complex scenarios (imdb$_{cc}$-imdb$_{cc3}$). The number of partitions entailed by imdb$_{cc}$, imdb$_{cc2}$, and imdb$_{cc3}$ 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 answer increases.

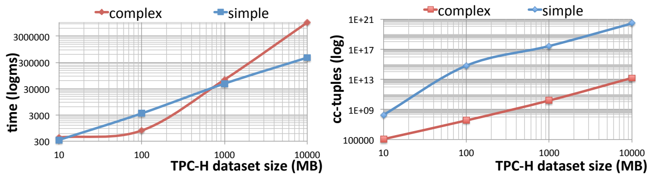
Increasing size of input instance. The last parameter we study is the input database size. To this end, we have created two scenarios, one with a simple and one with a complex Why-Not question t_c, and both using the same query Q_{tpch}. We run both scenarios for database sizes 10MB, 100MB, 1GB, and 10GB.

[4]http://www.tpc.org/tpch/



(a) simple conditions (b) complex conditions

Figure 8: *Ted*++ runtime w.r.t. number of conditions in t_c



(a) runtime (b) number of cc-tuples

Figure 9: *Ted*++ (a) runtime, and (b) number of cc-tuples for increasing database size, with complex and simple t_c

The simple t_c includes two inequality conditions, in order to be able to compute a satisfying number of cc-tuples. The complex t_c 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. This behavior is tightly coupled to the fact that the number of computed cc-tuples is augmenting proportionally to the increase of 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. Thus, the complex scenario loses its performance faster than the simple one in the big datasets.

In summary, our experiments have shown that *Ted*++ generates a more informative, useful and complete Why-Not answer than the state of the art. Moreover, *Ted*++ is either more efficient or comparable 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 a common laptop, with no special capabilities in memory or disk space, supports *Ted*++'s feasibility.

7 Conclusion and Outlook

This paper first introduced a novel representation of query-based explanations to Why-Not questions that ask why some data is not part of a result of a conjunctive query with inequalities. Our Why-Not answer takes the form of a polynomial that encodes all condition combinations of the query that are simultaneously responsible for pruning the missing answers from the result. These polynomials are shown to be more informative than Why-Not answers returned by previous algorithms. In addition, opposed to previous algorithms that may return a different result for any two equivalent query tree representations of the input query, we guarantee that the Why-Not answer polynomial is the same for a large set of equivalent query trees. To compute such polynomials, we first introduced the naive *Ted* algorithm that however is too inefficient to be of any practical use. Therefore, we presented a second, more efficient algorithm, namely *Ted*++. Our experimental evaluation showed that *Ted*++ is as efficient or more efficient than existing algorithms while providing more useful insights in its Why-Not answer to a developer. Also, we saw that

Ted++ scales well with various parameters, making it a practical solution as opposed to the naive *Ted*.

In the future, we plan to use the Why-Not answer polynomial to efficiently rewrite the input 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.

References

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

[2] A. V. Aho, Y. Sagiv, and J. D. Ullman. Equivalences among relational expressions. *SIAM Journal on Computing*, 8(2):218–246, 1979.

[3] S. S. Bhowmick, A. Sun, and B. Q. Truong. Why not, WINE? In *International World Wide Web Conference (WWW)*, pages 83–86, 2014.

[4] N. Bidoit, M. Herschel, and K. Tzompanaki. Immutably answering why-not questions for equivalent conjunctive queries. In *Workshop on Theory and Practice of Provenance (TAPP)*, 2014.

[5] N. Bidoit, M. Herschel, and K. Tzompanaki. Query-based why-not provenance with Nedexplain. In *International Conference on Extending Database Technology (EDBT)*, 2014.

[6] D. Calvanese, M. Ortiz, M. Simkus, and G. Stefanoni. Reasoning about explanations for negative query answers in dl-lite. *Journal on Artificial Intelligence Research (JAIR)*, 48:635–669, 2013.

[7] A. Chapman and H. V. Jagadish. Why not? In *International Conference on the Management of Data (SIGMOD)*, 2009.

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

[9] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS)*, 25(2), 2000.

[10] J. Danaparamita and W. Gatterbauer. QueryViz: helping users understand SQL queries and their patterns. In *International Conference on Extending Database Technology (EDBT)*, 2011.

[11] T. Grust and J. Rittinger. Observing sql queries in their natural habitat (preprint). *ACM Transactions on Database Systems*, 0(0), 2012.

[12] M. Hall. *Combinatorial theory*, volume 71. John Wiley & Sons, 1998.

[13] Z. He and E. Lo. Answering why-not questions on top-k queries. In *International Conference on Data Engineering (ICDE)*, 2012.

[14] M. Herschel. Wondering why data are missing from query results? ask conseil why-not. In *International Conference on Information and Knowledge Management (CIKM)*, 2013.

[15] M. Herschel and H. Eichelberger. The Nautilus Analyzer: understanding and debugging data transformations. In *International Conference on Information and Knowledge Management (CIKM)*, 2012.

[16] M. Herschel and M. A. Hernández. Explaining missing answers to SPJUA queries. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1), 2010.

[17] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1), 2008.

[18] T. Imieliński and J. Witold Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4), 1984.

[19] M. S. Islam, R. Zhou, and C. Liu. On answering why-not questions in reverse skyline queries. In *International Conference on Data Engineering (ICDE)*, 2013.

[20] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. SnipSuggest: Context-aware autocompletion for SQL. *Proceedings of the VLDB Endowment (PVLDB)*, 4(1), 2010.

[21] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, Jan. 1988.

[22] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *PVLDB*, 2011.

[23] D. Mottin, A. Marascu, S. B. Roy, G. Das, T. Palpanas, and Y. Velegrakis. A probabilistic optimization framework for the empty-answer problem. *Proceedings of the VLDB Endowment*, 6(14), 2013.

[24] A. Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *Proceedings of the VLDB (PVLDB)*, 4(12), 2011.

[25] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *SIGMOD Conference*, 2014.

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

[27] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems with negative provenance. In *ACM SIGCOMM 2014 Conference*, pages 383–394, 2014.

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399