Réponses manquantes : Débogage et Réparation de requêtes

Aikaterini Tzompanaki

2015SACLS223

UNIVERSITÉ PARIS-SUD
UNIVERSITÉ PARIS-SACLAY

ÉCOLE DOCTORALE INFORMATIQUE DE PARIS-SUD (ED 427)
Laboratoire de Recherche en Informatique (LRI)

DISCIPLINE *INFORMATIQUE*

THÈSE DE DOCTORAT

Soutenue le 14/12/2015 à Gif-sur-Yvette

par **AIKATERINI TZOMPANAKI**

Query Debugging and Fixing to Recover Missing Query Results

Directrice de thèse :	Nicole Bidoit	Professeur, Université Paris-Sud, France
Co-encadrante de thèse :	Melanie Herschel	Professeur, Université Stuttgart, Allemagne
Rapporteurs :	David Gross-Amblard	Professeur, Université Rennes 1 ISTIC, France
	Bertram Ludäscher	Professeur, Université Illinois, USA
	Yannis Velegrakis	Maître de conférences, Université Trento, Italie
Examinateurs :	Christine Froidevaux	Professeur, Université Paris-Sud, France
	Philippe Rigaux	Professeur, Conservatoire National des Arts et Métiers, France

Abstract

"Query Debugging and Fixing to Recover Missing Query Results"

Aikaterini Tzompanaki

With the increasing amount of available data and data transformations, typically specified by queries, the need to understand them also increases. "Why are there medicine books in my sales report?" or "Why are there *not* any database books?" For the first question we need to find the origins or *provenance* of the result tuples in the source data. However, reasoning about missing query results, specified by *Why-Not questions* as the latter previously mentioned, has not till recently received the attention it is worth of.

Why-Not questions can be answered by providing *explanations* for the missing tuples. These explanations identify why and how data pertinent to the missing tuples were not properly combined by the query. Essentially, the causes lie either in the input data (e.g., erroneous or incomplete data) or at the query level (e.g., a query operator like join).

Assuming that the source data contain all the necessary relevant information, we can identify the responsible query operators forming*query-based* explanations. This information can then be used to propose *query refinements* modifying the responsible operators of the initial query such that the refined query result contains the expected data. This thesis proposes a framework targeted towards SQL query debugging and fixing to recover missing query results based on query-based explanations and query refinements.

Our contribution to query debugging consist in two different approaches. The first one is a *tree-based approach*. First, we provide the formal framework around Why-Not questions, missing from the state-of-the-art. Then, we review in detail the state-of-the-art, showing how it probably leads to inaccurate explanations or fails to provide an explanation. We further propose the *NedExplain* algorithm that computes correct explanations for SPJA queries and unions there of, thus considering more operators (aggregation) than the state of the art. Finally, we experimentally show that *NedExplain* is better than the both in terms of time performance and explanation quality.

However, we show that the previous approach leads to explanations that differ for equivalent query trees, thus providing incomplete information about what is wrong with the query. We address this issue by introducing a more general notion of explanations, using *polynomials*. The polynomial captures all the combinations in which the query conditions should be fixed in order for the missing tuples to appear in the result. This method is targeted towards conjunctive queries with inequalities. We further propose two algorithms, *Ted* that naively interprets the definitions for polynomial explanations and the optimized *Ted++*. We show that *Ted* does not scale well w.r.t. the size of the database. On the other hand, *Ted++* is capable of efficiently computing the polynomial, relying on schema and data partitioning and advantageous replacement of expensive database evaluations by mathematical calculations. Finally, we experimentally evaluate the quality of the polynomial explanations and the efficiency of *Ted++*, including a comparative evaluation.

For query fixing we propose is a new approach for refining a query by leveraging polynomial explanations. Based on the input data we propose how to change the query conditions pinpointed by the explanations by adjusting the constant values of the selection conditions. In case of joins, we introduce a novel type of query refinements using outer joins. We further devise the techniques to compute query refinements in the *FixTed* algorithm, and discuss how our method has the potential to be more efficient and effective than the related work.

Finally, we have implemented both *Ted++* and *FixTed* in an system prototype. The query debugging and fixing platform, short *EFQ* allows users to interactively debug and fix their queries (conjunctive queries with inequalities) when having Why-Not questions.

Keywords: Why-Not questions, explanations, query debugging, query fixing, query refinement, data provenance, query verification

Résumé

“Réponses manquantes : Débogage et Réparation de requêtes”

Aikaterini Tzompanaki

La quantité croissante des données s’accompagne par l’augmentation du nombre de programmes de transformation de données, généralement des requêtes, et par la nécessité d’analyser et comprendre leurs résultats : (a) pourquoi telle réponse figure dans le résultat ? ou (b) pourquoi telle information n’y figure pas ? La première question demande de trouver l’origine ou *la provenance* des résultats dans la base, un problème très étudié depuis une 20taine d’années. Par contre, expliquer l’absence de réponses dans le résultat d’une requête est un problème peu exploré jusqu’à présent.

Répondre à une question Pourqoui-Pas consiste à fournir des explications quant à l’absence de réponses. Ces explications identifient pourquoi et comment les données pertinentes aux réponses manquantes sont absentes ou éliminées par la requête.

Notre travail suppose que la base de données n’est pas source d’erreur et donc cherche à fournir des explications fondées sur (les opérateurs de) la requête qui peut alors être raffinée ultérieurement en modifiant les opérateurs "fautifs". Cette thèse développe des outils formels et algorithmiques destinés au débogage et à la réparation de requêtes SQL afin de traiter des questions de type Pourqoui-Pas. Notre première contribution, inspirée par une étude critique de l’état de l’art, utilise un arbre de requête pour rechercher les opérateurs "fautifs". Elle permet de considérer une classe de requêtes incluant SPJA, l’union et l’agrégation. L’algorithme *NedExplain* développé dans ce cadre, a été validé formellement et expérimentalement. Il produit des explications de meilleure qualité tout en étant plus efficace que l’état de l’art.

L’approche précédente s’avère toutefois sensible au choix de l’arbre de requête utilisé pour rechercher les explications. Notre deuxième contribution réside en la proposition d’une notion plus générale d’explication sous forme de *polynôme* qui capture toutes les combinaisons de conditions devant être modifées pour que les réponses manquantes apparaissent dans le résultat. Cette méthode s’applique à la classe des requêtes conjonctives avec inégalités. Sur la base d’un premier algorithme naïf, *Ted*, ne passant pas à l’échelle, un deuxième algorithme, *Ted++*, a été soigneusement conçu pour éliminer entre autre les calculs itérés de sous-requêtes incluant des produits cartésien. Comme pour la première approche, une évaluation expérimentale a prouvé la qualité et l’efficacité de *Ted++*.

Concernant la réparation des requêtes, notre contribution réside dans l’exploitation des explications polynômes pour guider les modifications de la requête initiale ce qui permet la génération de raffinements plus pertinents. La réparation des jointures "fautives" est traitée de manière originale par des jointures externes. L’ensemble des techniques de réparation est mis en œuvre dans *FixTed* et permet ainsi une étude de performance et une étude comparative.

Enfin, *Ted++* et *FixTed* ont été assemblés dans une plate-forme pour le débogage et la réparation de requêtes relationnelles.

Keywords: Why-Not questions, explanations, query debugging, query fixing, query refinement, data provenance, query verification

Acknowledgments

I want to dedicate the first lines to my advisors, Nicole Bidoit and Melanie Herschel.

Nicole is the epitome of a responsible and caring advisor. From the very beginning she spent her valuable time to teach me the insights and formalities of databases, always patient with my questions and eager to explain even the smallest things. We have spent countless hours discussing ideas, hitting walls, and then coming up with solutions. Her door was always open for me whenever I needed her, while her deep knowledge and expert eye showed the right way to follow. However, if I said that Nicole was just an advisor to me, I would be underestimating the amount of respect and gratitude I feel towards her. In the good times she was genuinely happy for me and in the bad times no one could wish for a more supportive and encouraging mentor. For all that I will always be grateful.

Melanie has also been a valuable doctoral advisor for me. She introduced me to the exciting topic of data provenance, providing guidance and helpful advice. Her natural tendency to research and her critical mind in conjunction with her young age, have been an inspiration. Her attitude taught me that in life you should be bold, chasing what you want, insisting no matter the difficulties or rejections. I feel lucky that I collaborated with her and I owe her a big thank you!

I am also grateful to the reviewers of my thesis, David Gross Amblard, Bertram Ludaescher and Yannis Velegrakis for thoroughly reading my thesis and for their valuable comments. Further, I would like to thank Christine Froideveaux, and Philippe Rigaux for being part of my examining committee; I am very honored.

This PhD would not have been such a wonderful experience, without all the people with whom I had the pleasure to work and consort with in the office. Danai, Iwanna, Sejla, Damian, Jesus, Juan, Andres, Alessandro, Benjamin, Despoina, Zoi, Asterios, Paul, Alexandra, Raphael, Tushar, Fransesca, Benoit, Gianlucka, Yifan, Soudip ... thank you all for being part of my everyday life.

I am also honored to have been a member of the BD team (now LahDAK team) of LRI and to meet all these great people, Nathalie Pernell, Emmanuel Waller, Chantal Reynaud, Fatiha Sais, Sarah Cohen-Boulakia, Francois Goasdoue, Dario Colazzo, and so many others! Then, I would like to especially thank Ioana Manolescu for giving me the chance to be part of the OAK team of INRIA, and for providing me with concrete help when I needed it. I am also grateful to the University Paris Sud for financing my PhD and accepting my teaching services. Finally, I would like to thank the University of Stuttgart for the invites and the travel financial aid.

Being abroad for the first time was not easy and would have been unbearable, especially in the beginning, without all the good friends who I am blessed to have with me in Paris. Mary, Danai, Iwanna, Nelly, Boulouk, Mathiou, Dimitri, Fabien, Foivo, being with you helped me to not feel homesick, as you have been my family here in Paris. As such, I do not feel the need to say that I thank you but that I love you. Especially to my girls: Mary, thank you for being the one who knows what I want and what I think without the need to say a word. Danai, thank you for giving us your energy and being a friend on whom we can rely. Nelly, thank you for all the support and for being an inspiration when it comes to determination and passion for computer science. Iwanna, even though our friendship does not count many years, it counts for sure a lot in my heart.

Dimitra, Giohan, Charoula, being your friend so many years has helped me endure and enjoy every aspect of my life, since we were kids. Chryssa, Maria, Dina, Lenia thank you for always being there for me and lighting up my life! Even though we are far away, I always feel your love and your support.

Special thanks goes to my special one, Stamatis Zampetakis. Stamatis throughout the years has been my biggest supporter, my best friend, my loving companion. He endures without complaints my ups and downs, lifting me up whenever I fall, celebrating with me the happy moments. Being a great scientist and programmer himself, his help and support has not only been emotional but practical too. His opinion and comments on my work are always greatly valued, and in moments of coding crisis, he is always there to the rescue! Certainly, my life would be a worse place without him!

Last but not least, I would like to express my gratitude and love to my family: my parents Yannis and Athena, and my brother Spyros. They provided me with a peaceful environment while growing up, full of love, respect and commitment to each other. Through them I learned to love knowledge, and that it takes hard work and passion for whatever you do in order to succeed. Their love and support has always been unconditional and I hope that they realise how much I love them back.

To all of you, thank you, merci, ευχαριστώ!

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Asking questions is inherent to human nature. Why is the sky blue? Why did Homo sapiens survive through the centuries and Neanderthals did not? Why isn't the moon falling on the Earth? Why hasn't Leonardo di Caprio still won the Oscar? Except for the last one, the other questions are examples of how questions are the driving force for scientific progress. But as important as asking questions is, so important is to be able to understand the obtained answers. In this way, we are able to verify the sanity of both the information at hand and of the question itself.

Today, in many domains (for instance sciences, industry, or marketing), massive amounts of data are being generated and then processed to answer various types of questions (analytical, statistical, scientific, etc). After gathering raw data, this often involves combining and transforming these data through a number of possibly complex steps before obtaining the desired output data. One possibility to define such data transformations are declarative query languages. Indeed, these have the benefit of allowing developers to easily specify what the result of a transformation should be, without having to worry about the specific implementation that specifies how to obtain the result. On the downside, a declarative specification renders it more difficult to understand how result data were obtain, as there is no direct step-by-step walk-through of a query execution which developers are accustomed to from debuggers for procedural languages for instance.

Having said that, there is a need for debugging declarative queries, especially as data transformations become more complex and data volume increases. One means to provide debugging support on data obtained as a result to a query is to identify the original (raw) data that led to the result and to pinpoint the individual transformation steps these data did undergo. The field of data provenance (also known as data pedigree or data lineage) addresses these issues and thus enables users to validate result data output by a query. Indeed, provenance information can be used to verify that expected results were produced in the intended way and furthermore that they can be reproduced if necessary. Moreover, unexpected results can be traced back to possibly identify and fix the cause(s) that led to the unexpected result.

The provenance information mentioned above relates to data *present* in a query result. However, it is equally important to be able to understand why some expected data are *missing* from a query result. For instance, users may wonder “Why not any sales numbers from Asia?” in a sales report or “Why not any result tuples at all?" after trying to join data from multiple Web sources. These questions, to which we refer to as Why-Not questions, can be answered by providing *explanations* for the missing data. These explanations identify why and how data pertinent to the missing tuples were not properly combined by the query, answering for instance our Why-Not questions by “The source does not contain the region Asia" or “No data in one of the selected Web tables could find a join partner in another table". Essentially, the causes lie either in the input data or at data transformation steps (e.g., a query operator like join). Pinpointing these causes helps in initiating proper measures such as data cleaning or query fixing to recover the missing data.

To produce explanations based on erroneous data, we verify that the input data are sufficient to produce what we expect. In other words this can be translated as verifying that the database contains the ‘provenance’ information of what is missing. If the input data are not sufficient, the most relevant approach consists in computing candidate database updates that given the query would yield the missing tuples, an approach know as computing *instance-based* explanations. On the other side, if the input data contain all the necessary relevant information, we can identify why the query did not produce the missing tuples that we expected, i.e, which query operators are to blame for the missing tuples, an approach known as computing *query-based* explanations. This information can then be used to propose *query refinements* modifying the responsible operators of the initial query such that the refined query result contains the expected data.

Determining explanations and refinements as described above is a tedious, error-prone, and time consuming for a user who typically proceeds manually. More specifically, a user has to engage herself in a continuous trial and testing cycle of query debugging and fixing until her attempts yield the desired results. Taking into account the massive amounts of data to be handled and the growing complexity of transformations, explaining missing tuples manually easily becomes practically infeasible. Clearly, in this context, users would highly benefit from automatic support. Indeed, automatically computing explanations for missing tuples as described above reminds us of ‘classical’ debuggers and development tools put at programmers disposal for procedural programming languages. Debuggers for existing commercial database systems like Oracle[1], SQL Server[2] or MySQL/MariaDB[3], provide a debugger, yet with different focus: they either debug queries on the syntactic level, or they provide a guidance to debug stored procedures (i.e, programs encapsulating SQL queries) and functions by splitting the program into consecutive parts and providing intermediate results to the user. However, there is yet no system that can act as debugger for declarative queries and data transformations that can di-

[1] http://www.oracle.com/webfolder/technetwork/tutorials/obe/db/devdays2012/mod2_sqldev/mod2_sqldev.html

[2] https://technet.microsoft.com/en-us/library/cc646008(v=sql.105).aspx

[3] http://mydebugger.com/index.php

rectly indicate to the user the problematic part of their queries based on specific user demands. Indeed such a feature, could e added in existing debugging systems to enhance their effectiveness.

The overall goal of the Nautilus project [HE12] is to provide semi-automatic algorithms and tools for data transformation debugging, fixing, and testing. This thesis is set in the context of Nautilus and proposes a framework specifically targeted towards SQL query debugging and fixing to recover missing query results based on query-based explanations.

1.1 Contributions

As mentioned above, this thesis focuses on one particular problem in the more general context of declarative query debugging and fixing. That is, we concentrate on declarative queries specified in SQL and the particular debugging question of missing query results. In this setting, we further focus on computing and leveraging query-based explanations for debugging and fixing, respectively. That is, we assume the input data to be correct.

Let us now highlight the contributions of this thesis in the above context. In summary, this thesis is the first to provide a formal framework around Why-Not questions and relevant notions, which has been missing from prior work on this subject. Based on this solid theoretical foundation, we identify and overcome shortcomings of previous work such as result completeness or correctness. We propose two approaches to answer Why-Not questions with query-based explanations, namely the tree-based approach, referred to also as *NedExplain*, and the polynomial-based approach, referred to also as *Ted*. As we will see, these take an inherently different perspective of the query, resulting in different output properties, *Ted* being for instance the more general one. Exploiting the result of *Ted*, we further propose a novel approach towards fixing conjunctive queries with inequalities by building query refinements. These either relax selection conditions or transform joins to outer joins. Not only are we the first to consider query-based explanations when computing refinements, we are also the first to study in detail refinements with different join types. As there may be multiple possible refinements, we propose to rank these according to several criteria. After this brief contribution overview, we now describe our contributions in more detail.

Query debugging In supporting query debugging through query-based explanations, this thesis advanced the field of research in the following aspect.

1. **Tree-based approach** Our first solution for computing query-based explanation, named *NedExplain* [BHT14c] is inspired by the *Why-Not* algorithm [CJ09], which can be considered as the state-of-the-art algorithm. Like *Why-Not*, we follow a query tree-based approach. However, we significantly extend on this seminal work, as the below list of novel contributions clearly shows.

(a) *Formalization of query-based Why-Not provenance.* We provide a formalization of query-based explanations for Why-Not questions that was missing from prior work. Our model subsumes the concepts informally introduced previously and allows us to provably cover cases that were not properly captured by *Why-Not*. Besides properly setting definitions for the first time, our model also accommodates aggregate queries (i.e., select-project-join-aggregate queries, or SPJA queries for short) and unions thereof, aggregation not being considered by *Why-Not*.

(b) *The* NedExplain *Algorithm.* Based on the problem formalization, the *NedExplain* algorithm is designed to correctly compute query-based explanations given an SPJA query (or union thereof), a Why-Not question, a source database, and its associated schema. Here, a query-based explanation consists of operators of a specific algebraic query tree representation of the input query that are “responsible" for pruning data relevant to producing the missing results.

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

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

2. **Polynomial-based** Our second solution, described in *Ted* [BHT14a, BHT15a, BHT15d] is a novel idea to compute query-based explanations as *polynomials*, in a manner detached from query-trees. As we argue in our work, the query-tree based approach may lead to different and incomplete query-based explanations, when different trees are considered for the same query. Thus, unlike both *Why-Not* and *NedExplain*, the result of *Ted* is insensitive to the specific query tree representation of the input query and leads to complete and correct query-based explanations. Individual contributions are:

   (a) *Why-Not answer polynomial.* We introduce a novel representation of query-based explanations as a polynomial of query conditions, named **Why-Not answer polynomial**. The advantages of this representation are many. First, it captures all possible and complete query-based explanations, i.e., all the possible combinations of query conditions that prune relevant data from the result. Second, it provides an upper bound of the recovered missing answers by means of its coefficients. Third, Why-Not answer polynomials provide a unified framework to capture Why-Not answers under set, bag, and probabilistic data model semantics.

   Moreover, we show that isomorphic queries are equivalent w.r.t. the Why-Not answer polynomial for the same Why-Not question, which leads to showing that the explanations produced by *Ted* are more general the the ones produced by *NedExplain*. Finally, we show that in general equivalent queries do not correspond to the same Why-Not answer polynomial, given a Why-Not question, while we provide an approximate Why-Not answer polynomial for a query when a minimized version of it is available.

(b) *Naive* Ted *and optimized* Ted++*.* We provide two algorithms to produce Why-Not answer polynomials: a naive and straightforward algorithm called *Ted*, which is shown to be inefficient in practice, and the optimized *Ted++* algorithm that renders the solution practically interesting as well. *Ted++* relies on schema and data partitioning (allowing for a distributed computation) and advantageous replacement of expensive database evaluations by mathematical calculations and thus renders the solution more efficient.

(c) A comparative experimental evaluation shows that indeed, *Ted++* can efficiently compute Why-Not answer polynomials. Furthermore, experiments indicate that this form of explanation is capable to provide a wider and more accurate view of the potential problem in the query that led to missing results, compared to the query-based explanations returned by Why-Not and *NedExplain*. Finally, the experiments investigate the behaviour of *Ted++* w.r.t. different parameters (like database instance size, and query and Why-Not question size and conditions types), showing that it is an adequate algorithm for different scenarios.

Query Fixing In addition to contributing to debugging queries when facing missing results, we also propose a novel technique to make suggestions on how to fix the query. These suggestions are altered versions of the original query that we call refinements. In this context, our research is novel in the following aspects:

1. *Explanation-based refinements.* We propose a new approach towards fixing conjunctive queries with inequalities by building query refinements. This approach exploits the query-based explanations modelled as terms in a Why-Not answer polynomial. Starting the query refinement process using query-based explanations allows us to directly focus computation efforts on refinements altering the conditions identified as being too restrictive to make the missing results appear in the result.
2. *Refinements on selections and joins.* Our solution is the first to consider changing both selection and join conditions of the original query. A central question to answer here is how much to relax selections and to what joins can be changed. Essentially, the relaxation should ensure that the result of the refined query includes both the missing results and the original query result but at the same time avoid (too many) side-effects, i.e., further result tuples not satisfying a user's Why-Not question. Our solution relaxes the selection conditions based on a skyline selection of tuples relevant to producing the missing results (and not additional ones). This allows to efficiently compute refinements with a minimum number of changes (as far as the number of changes in the conditions is concerned), which we consider being "good" refinements as these are closest

to the user's initial intent. As for joins, we consider refining these to outer joins when possible. To the best of our knowledge we are the first to propose query refinements using outer joins. It actually turns out that the join graph in combination with the query-based explanation to a Why-Not question need to satisfy very specific properties for such a refinement to be possible.

3. FixTed *algorithm.* We provide the *FixTed* algorithm that computes query refinements for each query-based explanation of a Why-Not answer polynomial as described above. Among these, we are only interested in the best query refinements based on similarity and number of side-effects. To select these, we again rely on a skyline approach. The output of *FixTed* is a list of query refinements, ranked by similarity and/or number of side-effects.
4. *Prototypical implementation.* We have implemented *FixTed* [BHT15c], which together with *Ted++* build the main components of our *Explain and Fix Query (*EFQ*)* Platform. *EFQ* is provided as a web application and demonstrates that the proposed framework is feasible and convenient for a user that wants to know why certain results are missing and how they can be recovered.

1.2 Structure

This thesis is structured as follows:

Chapter 2 illustrates the background theory about relational databases and provides some contextualized introductory definitions in Section 2.1. Section 2.2 provides the definition of a Why-Not question and of central notions used in our solutions. Section 2.3 defines the input of our problem, before stating the two phases of the problem associated with query-based explanations and query refinements. Finally, Section 2.4 concludes the chapter.

Chapter 3 provides an overview of the most important related works on the field of Data provenance (Section 3.1) and Why-Not provenance (Section 3.2), answering Why and Why-Not questions respectively. We further delve into details for the publications on the Why-Not provenance field, placing the proposed algorithms of this thesis into the scenery and highlighting the differences with the state of the art. Section 3.3 concludes the chapter.

Chapter 4 discusses our proposal for the query debugging phase and consequently the query-based explanations. Section 4.1 provides the details of our first approach based on query trees and implemented in *NedExplain* algorithm. Section 4.2 introduces Why-Not answer polynomials and provides two algorithms *Ted* and *Ted++* to produce such kind of explanations. Moreover, here we discuss some conclusions about Why-Not answer polynomials about different database semantics and query equivalence. Section 4.3 concludes the chapter.

Chapter 5 presents our proposal for the query fixing phase by computing query refinements. We start by providing the motivation for the problem and the list of our contributions in Sections 5.1 and 5.2. In Section 5.3 we state the problem we address and provide some preliminary notions and definitions, used in this chapter. In Section 5.4 we define what are and how we compute the query refinements. Section 5.5 presents the *FixTed* algorithm that computes query refinements as defined in Section 5.4. Section 5.6 describes the *EFQ* platform, in which the implementation of the *EFQ* is incorporated. Finally, Section 5.7 concludes the chapter.

Chapter 6 concludes the thesis and outlines possible future directions.

Chapter 2

Preliminaries and Problem Definition

This chapter begins with an introduction to notions and concepts comprising the background on which this thesis problem is defined (Section 2.1). The contribution of this chapter is the formalization of the problem of Why-Not questions, provided in Section 2.2. More precisely, given a query and a database instance we define a Why-Not question, which we categorize as either *simple* or *complex*. Then, we define the relevant data in the database instance w.r.t. the Why-Not question which are called *compatible* data. In contradiction to related work that accounts only for simple Why-Not questions, our representation of compatible tuples also supports complex Why-Not questions. We close the chapter, by describing the problem statements addressed in this thesis in Section 2.3.

2.1 Relational data and query model

One of the most popular data models used worldwide is the relational model introduced by E.F.Codd in 1970 [Cod70]. The relational model provides a declarative way to specify data and queries over the data. By declarative we mean that the users do not need to describe in which way the data are structured internally or how the queries are evaluated. Instead, procedural means that the users provide the steps (i.e., the procedure) in which the data are created and the queries are executed.

In the following we describe the relational model for data and queries, adapted to the context of our problem.

A database schema $\mathcal{S}$ is a set of relation schema names. For example $\mathcal{S}=\{R, S, T\}$ is a database schema where R, S, T are relation schema names. A relation schema R is a set of attributes [1]. We assume each attribute of R qualified, i.e., of the form $R.A$. For any object O, $\mathcal{A}(O)$ denotes the set of attributes occurring in O, also referred to as the *type* of O. We assume that each database relation R has a special attribute $R.R\_Id$. For example, R consists of the attributes $\mathcal{A}(R)=\{R.A, R.B, R.R\_Id\}$. $dom(R.A)$ denotes the domain of $R.A$, i.e., the acceptable values that $R.A$ can take.

[1] For convenience, R refers to the schema and name of the relation.

For example, $dom(R.A) = \mathbb{Z}$. A *tuple* t is a list of attribute-value pairs of the form $(A_1{:}v_1, \ldots, A_n{:}v_n)$, where $\forall i{\in}[1,n]$: $v_i{\in}dom(A_i)$. For example, $t = (A{:}1, B{:}3)$ is a tuple. For conciseness, we may omit attribute names when they are clear from the context, i.e., write $(v_1, \ldots, v_n)$.

A tuple t over R is such that t and R have the same types, i.e., $\mathcal{A}(t){=}\mathcal{A}(R)$ and the value of each attribute in t is in the domain of the attribute, i.e., $t(R.A) \in dom(R.A)$. For example, $t = (A{:}1, B{:}3)$ is not a valid tuple for R because the attribute $R\_Id \in \mathcal{A}(R)$ is missing from t. An instance $\mathcal{I}_R$ over a relation R is a set of tuples over R. An instance $\mathcal{I}$ over a database schema $\mathcal{S}$, is a set of relation instances one for each $R \in \mathcal{S}$. The special attribute R_Id serves as the identifier of tuples in $\mathcal{I}_R$. $adom(R.A)$ denotes the active domain of $R.A$ in $\mathcal{I}_R$, i.e., the values used for $R.A$ in the instance $\mathcal{I}_R$. For example if $\mathcal{I}_R{=}\{(A{:}1, B{:}3, R\_Id{:}1)\}$ then $adom(R.A){=}\{1\}$. The definition of relation instance shows that we are considering databases under *set* semantics. Different database semantics include bag semantics, where relation instances are multi-sets of tuples, or probabilistic semantics, where each tuple may exist in a relation instance according to some probability. We revisit these semantics only in the dedicated Section 4.2.7.

Along with the relational data model, a relation query model is used to express queries against these data. The relational algebra [Cod72] proposed by E.F.Codd contains operators borrowed from the set theory, like set union, set difference and cartesian product, with some extra constraints. To apply union and difference over two relations, these have to be union-compatible, i.e., contain the same set of attributes, whereas the cartesian product demands that the relations contain disjoint attribute sets. Except for the set operators, the relational algebra includes projection, selection, rename and join operators. Aggregate functions [ÖÖM87] (*min, max, avg, count,* etc) extend the relational algebra. For example, $Q = \pi_{R.A}[\sigma_{R.B=S.B}[R \times S]]$ is a query expressed in relational algebra using cross-product and projection and selection operators. The most commonly implemented query language in relational database systems is the *Structured Query Language* (SQL [CB74]), initially based on the relational algebra described before. In this thesis we support SQL query expressions with the following syntax, where [ ] denotes optional and | denotes alternative form:

```
query::=            basic-query | basic-query ∪ query
basic-query::=      sel-clause [where-clause]
                    [GROUP BY col-name-list]
sel-clause::=       SELECT [DISTINCT] expr-list FROM table-name-list
expr-list::=        expr | expr-list, expr
table-name-list::=  table-name | table-name-list, table-name
col-name-list::=    col-name | col-name-list, col-name
where-clause::=     WHERE boolean
boolean::=          predicate | predicate AND boolean
predicate::=        comparator op comparand
comparator ::=      table-name.col-name
comparand ::=       table-name.col-name || constant
op ::=              > || < || = || ≥ || ≤ || ≠
expr::=             table-name.col-name || aggr-fn(table-name.col-name)
aggr-fn ::=         COUNT || SUM ||AVG || MIN || MAX
```

The semantics of an SQL query is that of the associated relational query described in the following discussion. For example, the semantics of the SQL query

```
SELECT DISTINCT R.A
FROM R,S
WHERE R.B=S.B
```

is given by the relational query $Q = \pi_{R.A}[\sigma_{R.B=S.B}[R \times S]]$.

To define the wider class of queries we are interested in, i.e., unions of aggregate queries, we start with the class of *conjunctive queries with inequalities.* A conjunctive query is a query in which only equality selections and joins are allowed; naturally, in a conjunctive query with inequalities, there may also exist inequalities and/or comparisons. To formally define this class of queries, we use the notion of *condition.* Assume available a set Var of variables and w.l.o.g. a unique domain $\mathcal{D}$. A condition *op* over Var 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 $\theta \in \{=, \neq, <, \leq, >, \geq\}$. The notation *op* used for condition is motivated by the fact that we may use it to refer to the operator to which a condition is associated (selection, join, etc). For a condition over a database schema $\mathcal{S}$ we assume that $Var = \mathcal{A}(\mathcal{S})$, i.e., the variables map to relation attributes. A condition over two relations is *complex*, otherwise it is *simple*, as in the following definition.

Definition 2.1.1. *(Simple/Complex condition) Let op be a condition, specified over the database schema $\mathcal{S}$. If the cardinality $\mid \mathcal{S} \mid$ is one, then op is a* simple *condition. If the cardinality $\mid \mathcal{S} \mid$ is two, then op is a* complex *condition.*

For example, $R.A \neq 3$ is a simple condition and $R.B = S.B$ is a complex one.

Definition 2.1.2 (Conjunctive query with inequalities)**.** *A conjunctive query with inequalities Q_{conj} is specified by a triple $(\mathcal{S}, \Gamma, C)$, where $\mathcal{S}$ is the input query schema, $\Gamma \subseteq \mathcal{A}(\mathcal{S})$ is the output query type, and C is a set of conditions over $\mathcal{A}(\mathcal{S})$. The semantics of Q is given by the relational algebra expression $\pi_\Gamma[\sigma_{\bigwedge\limits_{op \in C} op}[\times_{R \in \mathcal{S}}[R]]]$.*

For example, $Q{=}(\{R, S\}, \{R.A\}, \{R.B{=}S.B\})$ is a conjunctive query with semantics given by $Q{=}\pi_{R.A}[\sigma_{R.B=S.B}[R \times S]]$.

In what follows, when we refer to conjunctive queries we mean conjunctive queries with inequalities, unless stated otherwise. Note also, that given a query $Q{=}(\mathcal{S}, \Gamma, C)$ we use the notation $\mathcal{S}_Q$, Γ_Q, C_Q to refer to $\mathcal{S}$, Γ, C of Q when needed (for example when more than one queries are involved in the discussion).

Normally, two relation instances $\mathcal{I}_{R_1}$ and $\mathcal{I}_{R_2}$ may correspond to the same relation schema R. However, in our context we need to be able to distinguish among the attributes corresponding to $\mathcal{I}_{R_1}$ or $\mathcal{I}_{R_2}$, even if they (the attributes) have the same name. This is required to correctly deal with self-joins, as it will be made clear later. This is why we enforce that each relation has at most one occurrence in a query. In order to allow self-join we use renamed relation schema (à la SQL). For example, consider a database schema $\mathcal{S}^D = \{R\}$ and a query with a self-join between two instances of R. Then, the input schema of Q is $\mathcal{S}{=}\{R_1, R_2\}$, and not $\{R\}$, where R_1 and R_2 are renamings of R (in the definition bellow the renaming is formalized by η).

To establish the link between the input query schema and the database schema over which the query is specified and properly define a query over a database schema, we introduce the mapping η as follows.

Definition 2.1.3 (Query over a database). *A query over a database schema $\mathcal{S}^D$ is a pair (Q, η), where*

— *Q is a query with input schema $\mathcal{S}$, and*

— *η is a mapping from $\mathcal{S}$ to $\mathcal{S}^D$ such that $\forall R{\in}\mathcal{S}{:}R.A \in \mathcal{A}(R)$ if $\eta(R).A{\in}\eta(\mathcal{A}(R))$.*

Let $\mathcal{I}^D$ be a database instance over the database schema $\mathcal{S}^D$ and consider the instance $\mathcal{I}$ over the input schema $\mathcal{S}$ of Q, defined as $\mathcal{I}{=}\{\mathcal{I}_{|R} \mid R{\in}\mathcal{S}$ and $\mathcal{I}_{|R}{=}\mathcal{I}^D_{|\eta(R)}\}$. The evaluation of (Q, η) over $\mathcal{I}^D$ is defined as the evaluation of Q over $\mathcal{I}$.

Then, we define queries with aggregation over conjunctive queries (select-project-aggregation (SPJA) query) as follows.

Definition 2.1.4 (Aggregate query). *An aggregate query $\alpha_{G,F}$ over a conjunctive query $Q_{conj}{=}(\mathcal{S}, \Gamma, C)$ is specified by the quadruple $(\Gamma_{aggr}, Q_{conj}, G, F)$, where*

— *$G \subseteq \Gamma$ is the set of group by attributes,*

— *F is a list $f_i(W_i \to A^g_i)$ for $i{=}1, \dots, n$, where $f_i \in \{sum, count, avg, min, max\}$, A^g_i are new attributes (not in $\mathcal{S}$), and $W_i \subseteq \Gamma$, and*

— *$\Gamma_{aggr}{\subseteq}G \cup_{i=1,\dots,n} A^g_i$.*

The semantics of $\alpha_{G,F}$ is given by the (extended) relational algebra expression $\pi_{\Gamma_{aggr}}[\texttt{GROUPBY}_G\mathcal{F}_F([\pi_\Gamma[\sigma_{\bigwedge\limits_{op\in C} op}[\times_{R\in\mathcal{S}}[R]]]])]$, where F is the aggregation operator.

For example, consider the conjunctive query

$$Q{=}(\{R, S\}, \{R.A, R.B, S.C\}, \{R.B{=}S.B\})$$

The query $\alpha_{G,F}$=$(\pi_{\{R.A,sumC\}}, Q, G, F)$, where G=$\{R.A\}$ and F=$sum(S.C)$$\rightarrow$$sumC$, is an aggregate query. Its semantics is given by the extended relational query $\alpha_{G,F} = \pi_{R.A,sumC}[\texttt{GROUPBY}_{\{R.A\}}\mathcal{F}_{sum(S.C)\rightarrow sumC}([\sigma_{R.B=S.B}[R \times S]])]$.

Finally, we define unions of queries in a straightforward way. As relation schema attributes are qualified, two relation schemas always have disjoint types. To define union, we thus introduce renaming.

Definition 2.1.5 (Renaming ν). *Let Θ_1, Θ_2 and Θ_{new} be disjoint sets of attributes. A renaming ν w.r.t. Θ_1 and Θ_2 is a set of triples (A_1, A_2, A_{new}) s.t.*

— *$A_1$$\in$$\Theta_1$, $A_2$$\in$$\Theta_2$ and $A_{new}$$\in$$\Theta_{new}$*
— *ν defines the function $\Theta_i \rightarrow \Theta_{new}, i = 1, 2$*

Our notion of renaming captures both the specification of an association between attributes in Θ_1 and Θ_2, and a standard renaming.

Take for example the sets of attributes Θ_1=$\{A_1, A_2\}$, Θ_2=$\{B_1, B_2\}$, and Θ_{new}=$\{C_1, C_2\}$. ν=$\{(A_1, B_1, C_1), (A_2, B_2, C_2)\}$ is a renaming w.r.t. Θ_1 and Θ_2 mapping A_1 and B_1 to C_1, and A_2 and B_2 to C_2.

Definition 2.1.6. *(Union of queries) A union query Q_u is a quintuple $(\mathcal{S}, \nu, \Gamma, Q_1, Q_2)$ where*

1. *$\mathcal{S}$=$\mathcal{S}_1$$\cup$$\mathcal{S}_2$ is the input schema of Q_u*
2. *ν is a renaming w.r.t. $\Theta_1$$\subseteq$$\Gamma_1$ and $\Theta_2$$\subseteq$$\Gamma_2$*
3. *Γ=$\Theta_{new} \cup \Gamma_1 \setminus \Theta_1$ is the output schema of Q_u*
4. *Q_1 and Q_2 are two conjunctive or aggregate queries or*
5. *Q_1 and/or Q_2 is a union query*

The semantics of Q_u is given by the relational query: $\pi_\Gamma[[Q_1] \cup_\nu [Q_2]]$.

For example, assume the conjunctive queries
Q_1=$(\{R, S\}, \{R.A\}, \{R.B$=$S.B\})$ and
Q_2=$(\{P, T\}, \{P.A\}, \{P.C$=$T.C\})$.
Consider also the renaming ν=$(R.A, P.A, A)$.
Then $(\{R, S\}, \{A\}, Q_1, Q_2, \nu)$ is a union query. Its semantics is given by
$\pi_A[[\pi_{R.A}[\sigma_{R.B=S.B}[R \times S]]] \cup_\nu [\pi_{P.A}[\sigma_{P.C=T.C}[P \times T]]]$.

The result of a query Q (conjunctive, aggregate or union) over a database instance $\mathcal{I}$ over $\mathcal{S}$ is denoted by $Q[\mathcal{I}]$.

Note that the query classes defined here are not randomly chosen. In fact, our proposed methods and algorithms have been designed based on these classes. However, a more technical discussion is necessary in order to explain the limitation of the query language to these classes. A discussion on the class of queries chosen for each algorithm, as well as the possibly trivial extensions to larger classes, is provided in the respective chapters.

Alternative query representations For the purpose of the discussion in the subsequent chapters, we introduce two other ways to present queries: the tableau (related to [ASU79]) and the query tree representation.

In our setting, we use the query tableau representation for a conjunctive query Q with inequalities. The query tableau representation of Q (we may refer to this simply as query tableau Q) is a table having one row for each relation in the query input schema $\mathcal{S}$, one column for each attribute of each relation and one column with the query conditions set C.

The formal definition of a tableau query relies on v-tuples, which are tuples of variables. The variables of a v-tuple are similar in spirit to labelled nulls, used for instance in the context of data exchange [FKMP05]. In our context, the variables replace potential values of attributes.

Definition 2.1.7 (v-tuple). *Let Var be an enumerable set of variables and $\mathcal{S}$ a database schema. A v-tuple t_v over $\{A_1, \ldots, A_n\} \subseteq \mathcal{A}(\mathcal{S})$ is of the form $(A_1{:}x_1, \ldots, A_n{:}x_n)$, where $x_i \in Var$ for $i \in [1, n]$.*

Using the definition above, we can trivially extend v-tuples over relations. Next, $var(\cdot)$ is used to retrieve the set of variables from a structure, e.g., $var(A_1{:}x_1, \ldots, A_n{:}x_n)$ returns $\{x_1, \ldots, x_n\}$. We may also use a v-tuple t_v as a function mapping attributes to variables.

Now, we formally define query tableaux.

Definition 2.1.8 (Query tableau). *A query tableau Q over the input schema $\mathcal{S}$ is a triple $(\mathcal{B}, s, C')$ where*

1. *the body $\mathcal{B}$ is a mapping associating one v-tuple v to each $R{\in}\mathcal{S}$ s.t. $var(\mathcal{B}(R_1)){\cap}var(\mathcal{B}(R_2)){=}\emptyset$ for any distinct pair $R_1, R_2{\in}\mathcal{S}$,*
2. *the summary s is a set of distinguished variables s.t. $s{\subseteq}var(\mathcal{B})$, and*
3. *C' is a set of conditions over $var(\mathcal{B})$.*

Assume the set of variables Var as the co-domain of the bijection

$$h : \mathcal{A}(\mathcal{S}) \to Var$$

Then, the mapping of a query $Q{=}(\mathcal{S}, \Gamma, C)$ to its tableau representation $(s, \mathcal{B}, C')$ is straightforward:

— $\mathcal{S}$ is bijectively mapped to the body $\mathcal{B}$ of the tableau as discussed in the above definition

— Γ is bijectively mapped to the summary s, s.t. $\forall A \in \Gamma \exists x \in s : h(A) = x$

— C is bijectively mapped to C' as $h(C){=}C'$

Γ is mapped to the summary s and finally the It is obvious that the summary s is mapped to the projection set Γ, $\mathcal{B}$ is mapped to the the attributes in $\mathcal{S}$ and C maps to the set of conditions in Q.

For example, the query $Q{=}(\{R, S\}, \{R.A, R.B, S.C\}, \{R.B{=}S.B\})$ can be represented as the tableau query shown in Table 2.1. Each attribute from the database schema $\{R.A, R.B, S.B.S.C\}$ is mapped to a distinct variable x as follows:

$h(R.A) = x_1$
$h(R.B) = x_2$
$h(S.B) = x_3$
$h(S.C) = x_4$
Then, in Table 2.1 it is easy to see that $h(C) = C'$ and $h(\Gamma) = s$.

	$R.A$	$R.B$	$S.B$	$S.C$	C'
R	x_1	x_2			$x_2 = x_3$
S			x_3	x_4	$x_2 = x_3$
s	x_1	x_2	x_3		

Table 2.1: Example query tableau

For the purpose of the presentation of the class of SPJUA queries we use tree representation of queries. A tree representation of a query Q has one leaf for each relation of the schema $\mathcal{S}$ and the internal nodes are query operators (project, selection, join, aggregation, union). Note that selections and joins correspond to conditions in the set C of Q.

The following definition introduces queries in a classical manner and provides an immediate tree representation of a query. Note that to deal with natural join and union we use renamings as defined in Definition 2.1.5.

Definition 2.1.9 (Query tree)**.** *Let $\mathcal{S}$=$\{R_1, \ldots, R_n\}$ be a database schema. Then*

1. *$[R_i]$ is a query Q with input schema $\{R_i\}$ and output schema $\mathcal{A}(R_i)$, i∈$[1, n]$. $[R_i]$ has no proper subquery.*
2. *Let Q_1, Q_2 be queries with input schemas $\mathcal{S}_1$, $\mathcal{S}_2$, and output schema Γ_1, Γ_2 respectively. Assuming $\mathcal{S}_1$∩$\mathcal{S}_2$=∅:*
   - *$[Q_1] \bowtie_\nu [Q_2]$ is a query Q where ν is a renaming w.r.t. $\Theta_1 \subseteq \Gamma_1$ and $\Theta_2 \subseteq \Gamma_2$. The input schema of Q is $\mathcal{S}_1 \cup \mathcal{S}_2$.*
     Its output schema $\Theta_{new} \cup \Gamma_1 \cup \Gamma_2 \setminus \Theta_1 \setminus \Theta_2$.
   - *$\pi_W[Q_1]$ where $W \subseteq \Gamma_1$, is a query Q with input schema $\mathcal{S}_1$ and output schema W.*
   - *$\sigma_C[Q_1]$ where C is a condition over Γ_1, is a query Q with input schema $\mathcal{S}_1$ and output schema Γ_1.*
3. *Let Q_1 be a query according to (1) and (2), $G \subseteq \Gamma_1$ be a set of attributes and let F be a list $f_i(A_i \rightarrow A_i^g)$ for i=$1, \ldots, n$ where $f_i \in \{sum, count, avg, min, max\}$ and $A_i \in \Gamma_1$. Then, $G\mathcal{F}_F[Q_1]$ is a query Q with input schema $\mathcal{S}_1$ and output schema $G \cup_{i=1,\ldots,n} A_i^g$.*
4. *$[Q_1] \cup_\nu [Q_2]$ is a query Q where ν is a renaming w.r.t. $\Theta_1 \subseteq \Gamma_1$ and $\Theta_2 \subseteq \Gamma_2$ if Q_1 and Q_2 are queries according to (1), (2), (3), and (4). The input schema of Q is $\mathcal{S}_1 \cup \mathcal{S}_2$ and its output schema is $\Theta_{new} \cup \Gamma_1 \setminus \Theta_1$.*

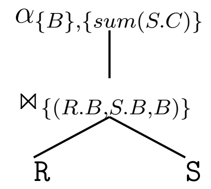


Figure 2.1: Example query tree

Consider for example the conjunctive query $Q{=}(\{R,S\},\{R.A\},\{R.B{=}S.B\})$. One possible query tree based on Definition 2.1.9 is $Q{=}\pi_{R.A}[Q_1]$, where $Q_1{=}[R]\bowtie_\nu[S]$ and $\nu{=}(R.B,S.B,B)$. Then, consider the aggregate query $\alpha_{G,F}{=}(\{B,sumC\},Q_1,G,F)$, where $Q_1{=}(\{R,S\},\mathcal{A}(\{R,S\}),\{R.B{=}S.B\})$, $G{=}\{B\}$ and $F{=}sum(S.C){\rightarrow}sumC$. The query tree $\alpha_{G,F}{=}\pi_{B,sumC}[B\mathcal{F}_{sum(S.C)}([Q_1])]$, where $Q_1{=}[R]\bowtie_\nu[S]$ and $\nu{=}(R.B,S.B,B)$ is an aggregate query tree, a representation of which can be seen in Figure 2.1.

Let us now focus on the output type $\Gamma_{aggr}{=}\{B,sumC\}$ of the aggregate query $\alpha_{G,F}$. Two attributes exist in Γ_{aggr}: B and $sumC$. Neither of these attributes exist in the input database schema, i.e., the input schema of Q_1. Indeed, the B has been introduced through the join renaming ν. $sumC$ is the named attribute corresponding to the output of the aggregate function *sum* over the attribute $S.C$. It is important to distinguish here these two different kinds of attributes that can be introduced in the output type of a query Q:

- the first one corresponds to renamed attributes because of a join or a union and is linked to a renaming function ν, and
- the second one corresponds to naming of attributes involved in aggregation functions.

The importance of this distinction will be clear in Section 4.1, when the *unrenaming* of output type attributes takes place.

2.2 Why-Not question and compatible data

In this section we discuss and define the most central notion, i.e., what a Why-Not question is and what it entails. As its name implies, a Why-Not question asks why the result of a query does not contain some tuples satisfying specific conditions posed by the user. As such, the Why-Not question consists in a set of conditions that are 'connected' to the original query in the sense that the conditions can only be specified over the query output schema. Apart from that, the Why-Not question does not depend on the conditions involved in the query. Abusively, we use here the terms Why-Not question and missing answer as synonyms, as adopted in the literature.

The fact that the Why-Not question is defined over the query output schema enables us to provide a unique definition of Why-Not questions regardless of the kind of query we are considering (conjunctive, aggregate or union). Thus, we can as well define a Why-Not question over the set of attributes Γ_Q. Note however, that for the case of aggregate queries we pose one restriction: one can only specify comparisons with constant values over attributes resulting from n aggregation functions (the set of aggregated attributes $\bigcup\limits_{i=1,\ldots,n} A_i^g$ in Definition 2.1.4). In other words, one cannot compare the result of an aggregation function with another attribute. Of course, for queries not involving aggregation the set of aggregated attributes is the emptyset.

Definition 2.2.1 (Why-Not question w.r.t. Q)**.** *Let Q be a query and let Γ_Q be its output schema. Let also $\bigcup\limits_{i=1,\ldots,n} A_i^g$ be the set of attributes introduced by the list of n aggregation functions F.*
A Why-Not question w.r.t. Q is the set of conditions WN s.t. if op is an operator then $op \in WN$ if

1. *$\mathcal{A}(op) \subseteq \Gamma_Q$ and,*
2. *if $\mathcal{A}(op) \cap \bigcup\limits_{i=1,\ldots,n} A_i^g \neq \emptyset$ then $\mathcal{A}(op)$ is a singleton.*

Consider for example the conjunctive query Q=$(\{R, S\}, \{R.A, R.B\}, \{R.B{=}S.B\})$. Then, WN=$\{R.A > 5\}$ is a Why-Not question w.r.t. Q.
Consider also the aggregate query $\alpha_{G,F}$=$(\{R.B, sumC\}, Q, \{R.B\}, sum(S.C){\to}S.C)$, where Q is the conjunctive query mentioned before. Then, WN=$\{R.B{=}6, sumC{>}4\}$ is a Why-Not question w.r.t. $\alpha_{G,F}$. However, WN=$\{R.B{=}sumC\}$ is not a Why-Not question because it includes a comparison of an aggregated attribute with another attribute (not a constant).

In the last example concerning aggregation, it is clear that a Why-Not question can be specified over attributes introduced by an aggregation function. However, we need to split the conditions in the Why-Not question into two subsets, based on whether they are specified over aggregated attributes or not. This distinction is important because only conditions over non-aggregated attributes will be later used to identify the *compatible* tuples. We postpone a more detailed discussion on the purpose of the distinction, when we focus on compatible tuples. So, we distinguish between:

1. the set WN_α=$\{op \mid op \in WN \text{ and } \mathcal{A}(op) \cap \bigcup\limits_{i=1,\ldots,n} A_i^g{\neq}\emptyset\}$, where $\bigcup\limits_{i=1,\ldots,n} A_i^g$ is the set of aggregated attributes resulting from n aggregation functions in an aggregated query, and

2. the set WN_{conj}=$WN \setminus WN_\alpha$.

For the previous example, it holds that WN_{conj}=$\{R.B{=}6\}$ and WN_α=$\{sumC{>}4\}$. Obviously, for conjunctive queries it holds that WN=WN_{conj}.

In our setting, a user is able to define one missing answer or a disjunction of missing answers. For example, given the query Q=$(\{R, S\}, \{R.A, R.B\}, \{R.B{=}S.B\})$, one could ask "*Why is there not any tuple with R.A>5 and R.B>8 or some other tuple with R.B=6 ?*". This is translated to two Why-Not questions:

$$\{R.A > 5, R.B > 8\}$$

or

$$\{R.B = 6\}$$

To express the disjunction of missing answers we define the *general* Why-Not question gWN as a set of Why-Not questions w.r.t. a query Q. For our example gWN=$\{\{R.A > 5, , R.B > 8\}, \{R.B = 6\}\}$ is a general Why-Not question.

Definition 2.2.2 (General Why-Not question w.r.t. Q)**.** *Let Q be a query. A* general *Why-Not question gWN w.r.t. Q is a set of Why-Not questions w.r.t. Q.*

As the Why-Not questions referred to in a general Why-Not question are independent from one another, we process and answer them independently. The results for each Why-Not question are then unified to provide the answer of the general Why-Not question. For this reason, in the following discussion we are considering one single Why-Not question as defined in Definition 2.2.1.

Following the notion of complex and simple conditions (Definition 2.1.1), complex and simple Why-Not questions are defined in a straightforward manner.

We further introduce the notion of *well defined* and *well founded* Why-Not question. Intuitively, a Why-Not question is meaningful if it addresses data not already returned by the query. To answer a Why-Not question with a query-based approach, we must identify the query conditions responsible for pruning out the relevant to the missing answer data from the source instance. Thus, only if we can identify such relevant data in the source instance, we are able to explain what has been wrong with the query conditions. If no relevant data exist in the source instance, then we should search for the problems in the source instance. This means that an instance-approach should be followed in order to indicate what data are missing from/should be updated in the source instance.

Definition 2.2.3 (Well defined Why-Not question)**.** *Let Q be a query and $\mathcal{I}$ a database instance over the schema $\mathcal{S}$ of Q. Then, a Why-Not question WN is said to be well defined if $Q[\mathcal{I}] \not\models \bigwedge_{op \in WN} op$.*

For example consider the query Q=$(\{R.A\}, \{R, S\}, \{R.B = S.B\})$ and its result $Q[\mathcal{I}]$=$\{(R.A{:}1), (R.A{:}2)\}$. The Why-Not question WN=$\{R.A > 1\}$ is a not well-defined because the result of Q contains the tuple $(R.A{:}2)$. On the other hand, WN=$\{R.A > 5\}$ is a well-defined Why-Not question because non of the result tuples satisfy the condition $R.A > 5$.

Definition 2.2.4 (Well founded Why-Not question)**.** *Let Q be a query and $\mathcal{I}$ be a database instance over the schema $\mathcal{S}$ of Q. Let also WN be a Why-Not question w.r.t. Q and consider its subset WN_{conj}. Then, WN is said to be well founded if it is well defined and $\mathcal{I} \models \bigwedge_{op \in WN_{conj}} op$.*

R

A	B	R_Id
1	2	Id_1
2	2	Id_2
7	3	Id_3

S

B	C	S_Id
2	2	Id_4
2	3	Id_5
1	1	Id_6

Figure 2.2: R and S relation instances

For example, let the active domain of $R.A$ w.r.t. $\mathcal{I}$ be $adom_{\mathcal{I}}(R.A)=\{1, 2, 7\}$. Then the well defined $WN=\{R.A > 5\}$ is well founded as well, because the value 7 in the active domain is greater than 5.

Note here that the conditions in WN_α are not taken into consideration in the definition of well-founded Why-Not question. This is because, as we already said,the conditions over aggregated attributes do not participate in identifying the relevant source data. This discussion needs more technical details and we revisit this issue in Section 4.1, when aggregation queries are considered.

In what follows we consider well-founded Why-Not questions.

To be able to obtain the missing answer in the query result, data from the input relation instances that satisfy the WN need to be combined by the query. The candidate data are what we call *compatible* tuples.

Previous works consider compatible tuples originating from different relations independently from one another, as discussed in Section 3.2.2. Take for example the database instance in Figure 2.2 and the query $Q=(\{R, S\}, \{R.A, S.C\}, \{R.B = S.B\})$ over this instance. Then, consider the Why-Not question $WN=\{R.A > 5, S.C = 3\}$. The tuple originating from $\mathcal{I}_R$ and relevant to WN is

$$(R.A{:}7, R.B{:}3, R\_Id{:}Id_3)$$

whereas the relevant tuple from $\mathcal{I}_S$ is

$$(S.B{:}2, S.C{:}3, S\_Id{:}Id_5)$$

So, the identified compatible tuples in this case are two, one from each relation.

Now, consider the Why-Not question $WN=\{R.A > S.C\}$. Following the previous logic, the compatible tuples from $\mathcal{I}_R$ are the tuples

$$(R.A{:}2, R.B{:}2, R\_Id{:}Id_2),\ (R.A{:}7, R.B{:}3, R\_Id{:}Id_3)$$

whereas from $\mathcal{I}_S$ all the tuples are compatible. However, consider the tuple from R identified by $R\_Id{:}Id_2$. This tuple should be considered compatible only in correlation with the tuple $S\_Id{:}Id_6$, for which it holds that $R.A > S.C$ (2>1). On the contrary, $R\_Id{:}Id_2$ should not be considered compatible in correlation with $S\_Id{:}Id_4$ because the condition $R.A > S.C$ is not satisfied for $R.A = 2$ and $S.C = 2$.

From the previous example, it is obvious that the definition of compatible tuples over one relation at a time, does not fit the case when complex Why-Not questions are considered. For this reason, we provide a new definition of compatible tuples as concatenation of tuples originating from the different relations in the database schema and satisfying the Why-Not question.

Definition 2.2.5 (Compatible tuple)**.** *Let Q be a query over the database schema $\mathcal{S}$, $\mathcal{I}$ be an instance over $\mathcal{S}$ and WN be a Why-Not question w.r.t. Q.*
Assume also the subset WN_{conj} of WN and consider the conjunctive query Q_{WN}=$(\mathcal{S}, \mathcal{A}(\mathcal{S}), WN_{conj})$.
The set CT=$Q_{WN}[\mathcal{I}]$ is the set of compatible tuples w.r.t. Q, $\mathcal{I}$, and WN.

Obviously, for a Why-Not question to be well founded, there should be at least one compatible tuple in $\mathcal{I}$.

Consider again the data and query of the previous example.For ease of reference, we refer to a tuple by its identifier attribute.
The compatible tuple w.r.t. WN=$\{R.A > 5, S.C = 3\}$ is $(R\_Id{:}Id_3, S\_Id{:}Id_5)$.
The compatible tuples w.r.t. WN=$\{R.A > S.C\}$ are

$$\begin{array}{c}(R\_Id{:}Id_2, S\_Id{:}Id_6)\\(R\_Id{:}Id_3, S\_Id{:}Id_4)\\(R\_Id{:}Id_3, S\_Id{:}Id_5)\\(R\_Id{:}Id_3, S\_Id{:}Id_6)\end{array}$$

It is easy to see that by Definition 2.2.1, the Why-Not question specifies a set of (missing-) tuples that the user expected to find in the query result and that these missing tuples have some common properties

1. their schema is the output query schema,
2. their values come from tuples in the database instance, and
3. they satisfy the conditions in the Why-Not question (more precisely in WN_{conj}).

Continuing the previous example, for the Why-Not question WN=$\{R.A{>}S.C\}$ the missing tuples are

$$\begin{array}{c}(R.A{:}2, S.C{:}1)\\(R.A{:}7, S.C{:}2)\\(R.A{:}7, S.C{:}3)\\(R.A{:}7, S.C{:}1)\end{array}$$

The compatible tuples directly provide the ‘provenance’ of the missing tuples, as the missing tuples can be obtained from the compatible ones by projection on the attributes of the output query schema. This can be easily observed in the previous example.

Intuitively, the reason that makes a tuple missing is that its associated compatible tuples were pruned out by the query conditions. In the next chapter, we show that answering a Why-Not question amounts to identifying the pruning query conditions for the compatible tuples in CT.

The schema of a compatible tuple τ consists of all the attributes in the schema $\mathcal{S}$, as seen by the signature of the query Q_{WN} in Definition 2.2.5. For the purpose of our study, we are now going to partition the schema of compatible tuples to split these tuples into partial tuples such that

— each partial tuple s is associated with a subset of conditions WN_s of the Why-Not question WN; these conditions are those using the attributes of the partial tuple s in their formula, and moreover

— two partial tuples s_1 and s_2 of a same compatible tuple do not share any (associated) conditions (in the sense explained above) meaning that checking conditions in WN can be decomposed in checking WN_{s_i} on s_i, for i=1, 2 independently.

Partitioning the schema of compatible tuples (partitioning S) is thus closely related to partitioning the set of conditions in WN and thus also of the query Q_{WN}. The purpose of Definition 2.2.6 is to state what is a valid partitioning of $\mathcal{S}$.

Definition 2.2.6. *(Valid Partitioning of $\mathcal{S}$). Consider the set of conditions WN and its subset WN_{conj}. The partitioning $\mathcal{P} = \{Part_1, \ldots, Part_k\}$ of the database schema $\mathcal{S}$ is* valid *if each $Part_i$, $i\in\{1,\ldots,k\}$ is minimal w.r.t. the following property:*

if $R\in Part_i$ and $R'\in\mathcal{S}$ s.t.

$\exists op\in WN_{conj}$ s.t.

$\mathcal{A}(op)\cap\mathcal{A}(R')\neq\emptyset$ and

$\mathcal{A}(op)\cap\mathcal{A}(R)\neq\emptyset$

then $R'\in Part_i$.

For example, if $\mathcal{S}$=$\{R, S\}$ and WN=$\{R.A > 5, S.C = 3\}$, then the valid partitioning of $\mathcal{S}$ w.r.t. WN is $\mathcal{P}$=$\{\{R\}, \{S\}\}$.
If we consider the Why-Not question WN=$\{R.A>S.C\}$, then the valid partitioning of $\mathcal{S}$ w.r.t. WN is $\mathcal{P}$=$\{\{R, S\}\}$.

The set $CT_{|Part}$ of *partial* compatible tuples w.r.t. $Part\in\mathcal{P}$ is obtained by evaluating the query $(Part, \mathcal{A}(Part), WN_{|Part})$ over $\mathcal{I}_{|Part}$. $WN_{|Part}$ and $\mathcal{I}_{|Part}$ denote the restriction of WN and $\mathcal{I}$ over the relations in $Part$, respectively.

It is easy to prove that the valid partitioning of $\mathcal{S}$ is unique and that the set CT can be computed from the sets $CT_{|Parti}$ of partial compatible tuples.

Lemma 2.2.1. *Let $\mathcal{P}$ be the valid partitioning of $\mathcal{S}$. Then,*

$$CT = \times_{Part_i\in\mathcal{P}} [CT_{|Part_i}]$$

Proof. Let Q be a query over $\mathcal{S} = \{R_1, \ldots, R_n\}$.
Let WN be a Why-Not question w.r.t. Q and assume known the subset WN_{conj}.
Let $\mathcal{I}$ be an instance over the database schema $\mathcal{S}$.
Let $\mathcal{P}$=$\{Part_1, \ldots, Part_k\}$ be the valid partitioning of $\mathcal{S}$.

Let $WN_{conj\ |Part}$ be the restriction of the Why-Not question over the relations in $Part$.

By Definitions 2.1.2, 2.2.5 , the set of compatible tuples CT is the result of the query

$$Q_{WN}{=}\sigma_{\bigwedge\limits_{op\in WN_{conj}} op}[R_1 \times \ldots \times R_n]$$

evaluated over $\mathcal{I}$. Q_{WN} can be re-written as (equivalent to pushing selections down)

$$\begin{aligned} Q_{WN} &= [\sigma_{\bigwedge\limits_{op\in WN_{conj\ |Part_1}} op}[\times_{R\in Part_1} R]] \times \ldots \times [\sigma_{\bigwedge\limits_{op\in WN_{conj\ |Part_n}} op}[\times_{R\in Part_n} R]] \\ &= Q_{WN_{|Part_1}} \times \ldots \times Q_{WN_{|Part_n}} \end{aligned} \tag{1}$$

Note that we assume that $\sigma_{\bigwedge\limits_{op\in\emptyset}}[R]{=}[R]$.

By Definition 2.2.6, it holds that each relation belongs exactly to one partition. Assume $CT_{|Part} = Q_{WN_{|Part}}[\mathcal{I}_{|Part}]$. Then, it follows from Equation (1) that

$$CT = CT_{|Part_1} \times \ldots \times CT_{|Part_n}$$

□

2.3 Problem statement

The main objective of this thesis is to provide aid to SQL developers in the context of missing answers from query results. The provided aid is two-fold: firstly, we automatically debug the query by identifying the culprit query conditions and secondly, we provide alternative query refinements that fix the problem of missing answers.

For both the problems we consider given (by the user) a number of input parameters that we package under a common name.

Explanation Scenario: An explanation scenario Δ is specified by the quadruple $(\mathcal{S}, \mathcal{I}, Q, WN)$, where

1. $\mathcal{S}$ is a database schema,
2. $\mathcal{I}$ is a database instance over $\mathcal{S}$,
3. Q is a query with input schema $\mathcal{S}$, and
4. WN is a well defined Why-Not question w.r.t. Q and $\mathcal{I}$.

When a user is confronted with query results that do not contain certain expected tuples, she may want to know if the query that produced these results is not correct and why it is not correct. This is what we call the *query debugging* problem. To explain why the query cannot produce the expected tuples, our solution identifies which query conditions are contradicting the user's Why-Not question and if properly repaired, the missing answers would appear in the result. From another perspective, our first problem targets query-based explanations to Why-Not questions, defined as follows.

Definition 2.3.1 (Query-based explanation). *Given an explanation scenario Δ, a query-based explanation is a minimal set of conditions*
$QBE \subseteq C_Q$ s.t. for the query Q_{QBE}=$(\mathcal{S}_Q, \Gamma_Q, C_Q \setminus QBE)$ it holds that
$Q_{QBE}[\mathcal{I}] \models WN$.

Problem Statement 2.3.1 (Query debugging). *Given an explanation scenario Δ, how can we effectively and efficiently generate query-based explanations QBE to solve Δ?*

Furthermore, it remains an open question how to present the query operators that are part of query-based explanations to make them most useful to the developer. Is it better to return only one candidate and if so, which one ([CJ09] opted for this approach) or is it better to return a (ranked) list of all potential combinations of responsible operators? Moreover, should data examples be returned along with the operators? This thesis defines and analyzes two alternative solutions, devises algorithms computing these and also evaluates them experimentally. More details are provided in Chapter 4.

The result of the query debugging supplies the user with useful knowledge for subsequently fixing the query, as she knows exactly which combinations of conditions to target. To help the user in this process, the second problem addressed in this thesis asks for alternative ways of refining the query in order to be able to return satisfying results w.r.t. the missing answer. The second problem we address is about computing query refinements, defined as follows.

Definition 2.3.2 (Query-refinement). *Given an explanation scenario Δ, a query refinement is a query*
Q' s.t. $Q[\mathcal{I}] \subset Q'[\mathcal{I}]$ and $Q'[\mathcal{I}] \models WN$.

Note that in the previous definition, the class of queries in the input and in the output may differ. Even though we are considering that the user provides a conjunctive query to be refined, it may be the case that the proposed refinement could contain outer joins instead of inner joins. More details are given in the dedicated Chapter 5.

Problem Statement 2.3.2 (Query fixing). *Given an explanation scenario Δ, how can we efficiently compute a set of useful query refinements?*

What is a useful query refinement is a rather subjective matter, and in general it would be hard to formally define the notion of usefulness. A priori, and as per the definition of a query refinement, the refined query should satisfy the user in terms of obtained results. However, in order for the query to be useful it should reflect the user's intentions and style as much as possible. This translates both on the level of the syntax of the refined query and of the obtained query results. In order to maximize the possibilities that the user finds useful the proposed refinements, we provide in this thesis alternatives with respect to the changes we make in the query and to the results that are returned. How (in what order) to display the obtained refined queries to the user is also subject to research, as the alternatives may abound. More details are given in Chapter 5, where our solution is described.

2.4 Summary

In this chapter we presented the background on which this thesis is based, that is the relational data model and queries. Then, we provided the introductory definitions for establishing the Why-Not questions problem, including novel definitions for compatible data and Why-Not questions. Finally, we stated the exact problems addressed in this thesis, about computing query-based explanations and query refinements, to aid users debug and fix their queries respectively.

The reader can find in Table 2.2 a quick reference to notations frequently used throughout the thesis.

Notation	Meaning
$\mathcal{S}$	database schema
$\mathcal{A}(x)$	the set of attributes in x
$\mathcal{I}$	a database instance
$\mathcal{I}_R$	a relation instance over the relation R
op	condition
Q	query
C_Q	set of conditions of Q
Γ_Q	set of projected attributes of Q
ν	renaming
$\alpha_{G,F}$	aggregation operator, where G denotes the group-by attributes and F the set of aggregation functions
WN	Why-Not question
gWN	general Why-Not question
CT	set of compatible tuples
$\mathcal{P}$	valid partitioning of a database schema
$Part$	connected component of $\mathcal{P}$
Δ	explanation scenario

Table 2.2: Notations table

Chapter 3

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.). This has led to the necessity of providing algorithms and tools to better understand and control the behavior and semantics of developed data transformations, and various solutions have been proposed so far, including data lineage [CWW00] and more generally data provenance [CCT09], (sub-query) result inspection and explanation [GKRS11, RS14], query conditions relaxation [MMR$^+$13] or query repairing in data integration systems [BFS00]. Also tools have been proposed for simplifying the specification of complex data transformations [KKBS10, NJ11] and recently [BMNT15] has proposed a system for data cleaning in case of erroneous but also missing data using provenance and Why-Not questions to guide the cleaning task. In the same spirit, cooperative database systems propose to not only output query results but also escort these with useful explanations like for the case of empty results[God97].

Further approaches useful for query debugging and relevant to data provenance include methods for weighting and ranking the possibly large provenance, e.g., based on causality and responsibility [MGMS11] or methods to automatically generate test data given a query and a desired output [BKL07, OCS09]. The latter are particularly valuable for instance when the source data are not accessible or incomplete. Further support in verifying transformation behavior and semantics can be obtained through sub-query result visualization [DG11], or tools that simplify the specification of complex data transformations [KKBS10, NJ11].

Whereas the methods mentioned above focus on relational queries, other approaches have considered more complex data transformations, where in the worst case, individual manipulations use black-box functions. Clearly, understanding the results of data transformation is essential in this context, and first solutions to incorporate data provenance in scientific workflows [ADD$^+$11] or Map-Reduce workflows [PIW11] have been proposed. The need of capturing and managing the possibly large provenance of data obtained in scientific workflows is evident in various systems, like Taverna [OAF$^+$04], Kepler [LAB$^+$06], PBase [CVKL$^+$14], etc. We further observe that fundamental ideas underlying data provenance techniques have also been considered for Datalog [GKT07, KLS12], logic programming [DAA13], XML [FGT08], and SPARQL [DAA12, GKCF13].

The work presented in this thesis is interested in the observation and understanding of relational query results. This in general may concern data existing in the query results and data missing from the query results. Thus, we review works in this context, distinguishing them into two categories considering:

1. Data provenance (Section 3.1). Here, the problem lies on understanding the origins of data existing in a query result.
2. Why-Not provenance (Section 3.2). Here, the problem lies on understanding the reasons that caused data to be absent from a query result.

This criteria can also be seen as the type of questions each work aims to answer, i.e., 'Why' or 'Why-Not' questions.

As introduced in Chapter 2, in this thesis we target our research to explaining why data are missing from the query result. Thus, we provide here a more thorough study for works in the 'Why-Not' provenance category. We group the existing works based on the type of explanation they return in order to explain why data are missing from a query result. The type of the explanation is determined by the primitive elements used in the explanation, for example, tuples or query conditions. So, the following five categories are distinguished and are in detail described in the next sections:

1. instance-based explanations, when they consist of tuples
2. query-based explanations, when they consist of conditions of a given query
3. hybrid explanations, when they consist of both conditions and tuples
4. ontology-based explanations, when they consist of concepts of an ontology
5. query refinements, when they consist of queries

From these five categories, we further specialize our work to query-based explanations and query refinements for missing answers, as we discussed in the problem definition section (Section 2.3) of Chapter 2. Thus, we are particularly emphasizing to works described in the respective sections (Section 3.2.2 and 3.2.5).

3.1 Data Provenance

The *data provenance* field, which started with the introduction of *data lineage* in relational databases, counts more than one decade of study [WS97]. As surveyed in [CCT09], *data provenance* research focuses on explaining data present in a query result and may be categorized in three forms based on *where* [BKT01] the data were copied from (exactly what attributes of which tuples), *why* [BKT01, CWW00] a query answer was produced (i.e., based on what source data), and *how* [GKT07] data were manipulated by the query to produce the result data in question.

Note that our work is interested in *fine-grained* provenance at the level of individual tuples. On the other side, *coarse-grained* provenance is mostly applicable to workflows and records the sequence of steps taken in a workflow system to derive the dataset and may even record involved software and hardware ([BT07]). Typically, coarse-grained provenance is documented in metadata.

The list of works in data provenance is long, documented in surveys and tutorials (like [CCT09, BT07]). Since the work in this thesis is indirectly connected to the data provenance problem, we restrict the discussion here to the works from the categories of why and how provenance that we use in our work and to which we correlate.

Why Provenance

In [CW00, CWW00] the authors defined the *lineage (or derivation)* - a form of why provenance - of data in the output of (materialized) relational views in a warehouse environment. Intuitively, the lineage of a result tuple t captures the source tuples that were combined in the view and resulted in t. The specification of the views considered in this context are select-project-join-aggregation-union-set difference (SPJUAN) queries. The definitions consider views as operator trees evaluated bottom up. Then, lineage is defined firstly for one operator and then recursively for the rest of the view. Since tuple lineage is a notion exploited in this thesis, we reproduce here the definition provided in [CW00, CWW00].

Definition 3.1.1. *(Tuple Lineage for one operator) Let op be any relational operator over the relation instances $\mathcal{I}_{R_1}, \ldots, \mathcal{I}_{R_n}$, and let $\mathcal{I}_R = op(\mathcal{I}_{R_1}, \ldots, \mathcal{I}_{R_n})$ be the relation instance that results from applying op to these instances. Given a tuple $t \in \mathcal{I}_R$ we define t's lineage in $\mathcal{I}_{R_1}, \ldots, \mathcal{I}_{R_n}$ according to Op to be*

$$op^{-1}_{<\mathcal{I}_{R_1},\ldots,\mathcal{I}_{R_n}>}(t) = < \mathcal{I}^*_{R_1}, \ldots, \mathcal{I}^*_{R_n} >$$

where $\mathcal{I}^*_{R_1}, \ldots, \mathcal{I}^*_{R_n}$ are maximal *subsets of $\mathcal{I}_{R_1}, \ldots, \mathcal{I}_{R_n}$ such that:*

1. $op(\mathcal{I}^*_{R_1}, \ldots, \mathcal{I}^*_{R_n}) = \{t\}$
2. $\forall \mathcal{I}^*_{R_i} : \forall t^* \in \mathcal{I}^*_{R_i} : op(\mathcal{I}^*_{R_1}, \ldots, t^*, \ldots, \mathcal{I}^*_{R_n}) \neq \emptyset$

Also, we say that $op^{-1}_{\mathcal{I}_{R_i}}(t) = \mathcal{I}^*_{R_i}$ is t's lineage in $\mathcal{I}_{R_i}$, and each tuple t^* in $\mathcal{I}^*_{R_i}$ contributes to t, for $i = 1 \ldots n$.

To make the definition clear, the first point declares that the operator *op* when executed over the subsets $\mathcal{I}^*_{R_i}$ produces exactly the tuple t. Thus, the lineage of a tuple t given an operator *op* cannot result in a different tuple. The second point of the definition means that only tuples contributing to the generation of the tuple t, by *op*, exist in the lineage of t. Thus, there are not 'irrelevant' tuples in the considered subsets. For example, tuples not satisfying a selection condition will not appear in the lineage of any result tuple. By defining the $\mathcal{I}^*_{R_i}$s to be the maximal subsets that satisfy requirements (1) and (2), it is ensured that the lineage contains exactly all the tuples that contribute to t.

In this thesis, we use the definition of lineage for tuples resulting from projection, selection and join relational operators. To further clarify lineage, consider the following example.

R

A	B	R_Id
1	2	Id_1
2	2	Id_2
7	3	Id_3

S

B	C	S_Id
2	2	Id_4
2	3	Id_5
1	1	Id_6

Figure 3.1: Relation instances for Example 3.1.1

Example 3.1.1. *Consider the instances of the relations R and S in Figure 3.1. If $op : \sigma_{A=1}[R]$, then the result tuple is $\{t\}$=$op(\mathcal{I}_R)$=$(R.A{:}1, R.B{:}2, R\_Id{:}Id_1)$. The lineage of t is $op^{-1}_{<\mathcal{I}_R>}(t)$= $< \{(R.A{:}1, R.B{:}2, R\_Id{:}Id_1)\} >$. Property 1 holds because the execution of the operator over the instance $\{(R.A{:}1, R.B{:}2, R\_Id{:}Id_1)\}$ yields exactly t. This means that the tuples in the lineage of t generate exactly t. Property 2 holds, because there is only one tuple in $\mathcal{I}^*_R$, from which the selection generates the tuple t. This means that every tuple in the lineage of t contributes to generation of t.*

Similarly, if op:$[R] \bowtie_{R.A=S.C} [S]$, one of the result tuples is t=$(R.A{:}1, R.B{:}2, R\_Id{:}Id_1, S.B{:}1, S.C{:}1, S\_Id{:}Id_6)$∈$op(\mathcal{I}_R, \mathcal{I}_S)$. The lineage of t is $op^{-1}_{<\mathcal{I}_R>,<\mathcal{I}_S>}(t)$= $< \{(R.A{:}1, R.B{:}2, R\_Id{:}Id_1)\}, \{(S.B{:}1, S.C{:}1, S\_Id{:}Id_6)\} >$.

How Provenance

In [GKT07] the authors propose a semiring representation of the provenance of tuples in the result of a query. They show that querying annotated databases, like probabilistic, incomplete or databases under bag semantics, are all instances of the same positive algebra computation algorithm over K-relations that lead to a more representative form of provenance using polynomials, a special class of semirings. Indeed, they state that by annotating each result tuple with its polynomial-like provenance, they provide a more comprehensive understanding of *how* a tuple was produced, and not only why it was produced as the tuple's lineage [CWW00] suggests.

For an example, consider a tuple t resulting from a join over two source tuples t_1 and t_2. Then, t is annotated with the polynomial t_1t_2 showing that t is derived by combining t_1 and t_2. If t is the result of a union over t_1 and t_2 then it would be annotated with $t_1 + t_2$ showing that t can be derived either from t_1 or from t_2. The lineage of t in both cases is $< \{t_1\}, \{t_2\} >$, which is not as informative as the polynomial.

Moreover, the authors in [GKT07] study provenance semirings for datalog queries over incomplete and probabilistic databases, through formal power series.

[ADT11b] extends the framework to queries with aggregation and difference. Here the framework of semiring annotations built with elements from a K-relation, is extended to accommodate more operators for aggregation. This results in introducing a *semimodule* algebraic structure over the semiring considering a commutative monoid $(M, +_M, 0_M)$, where the elements of M are elements of the database domain (i.e., attribute values). This allows for capturing not only the provenance of result tuples, but also the provenance of *values* resulting from the execution of an aggregation query.

Example 3.1.2. *Consider the aggregation query (expressed in relational algebra)*

$$Q{=}R.B\mathcal{F}_{R.A\rightarrow sumA}[R]$$

The result over the instance of R in Figure 3.1, is

$$Q[\mathcal{I}_R]{=}\{(R.B{:}2, sumA{:}3), (R.B{:}3, sumA{:}7)\}$$

The provenance of the attribute sumA in the tuple $(R.B{:}2, sumA{:}3)$ is described by the expression $Id_1 \otimes 1$ sum $Id_2 \otimes 2$. In the case of bag semantics, if Id_1 is mapped to 2 and Id_2 is mapped to 3 then, the previous expression is evaluated to 2 sum $6 = 8$, which represents the value of sumA.

3.2 Why-Not Provenance

Now, we focus our attention on works answering Why-Not questions and providing explanations about the encountered problems that lead to missing-answers. If we consider a scenario $\Delta{=}(\mathcal{S}, \mathcal{I}, Q, WN)$,as described in Notation 2.3, each reviewed algorithm proposes to explain such a scenario by either debugging or proposing modifications to one or more of the following parameters: (i) the underlying database ($\mathcal{S}$ or $\mathcal{I}$), (ii) the input query (Q), and (iii) the specified Why-Not question (WN).

As already said, we distinguish among five categories for Why-Not provenance, grouping algorithms computing (i) instance-based, (ii) query-based, (iii) hybrid, (iv) ontology-based, and (v) refinement-based explanations.

The problem statements we defined for this thesis in Section 2.3, page 22, are about query debugging and query fixing. Thus, the proposed solutions fall into the category of query-based explanations and query refinements. We consider all other categories as orthogonal to these two categories. Indeed, some explanations are more appropriate than others in different applications or contexts. For example, in incomplete databases or for data cleaning tasks, when the source database is not completely trusted, instance-based explanations are important for curating/completing the data. However, when data is trusted, such kind of data modifications are not acceptable and one would rather opt for query-based explanations. Then, when data can also be accessed and queried through concepts of an ontology, ontology-based explanations seem to be well-fit.

Table 3.1 provides a structured overview of the works in the Why-Not provenance field. These works are categorized according to the type of explanation they generate and the table further reports the format of the generated explanation, the class of query and Why-Not question (simple or complex, see Definition 2.1.1) supported by each algorithm.

Table 3.1: Algorithms for answering Why-Not questions

Algorithm	**Why-Not question**	**Explanation format**	**Query**
Instance-based explanations			
NA [HCDN08]	simple	source table edits	SPJ
Artemis [HH10]	complex	source table edits	SPJUA
Meliou *et. al.* [MGMS11]	simple	causes (tuples) and responsibility	SPJ
Calvanese *et. al.* [COSS13]	simple	additions to ABox	instance & conj. queries over DL-Lite ontology
PGames [KLZ13, RKL14]	simple	source table edits	SPJUN
Roy and Suciu [RS14]	higher/lower	attribute-value pairs and degree	SPJA
Query-based explanations			
Why-Not [CJ09]	simple	query operators	SPJU
NedExplain [BHT14c]	simple	query operators	SPJUA
Ted [BHT14a] **Ted++** [BHT15a]	complex	polynomial	SPJU
Hybrid explanations			
Conseil [Her13, Her15]	simple	source table edits + query operators	SPJAN
Ontology-based explanations			
Cate *et. al.* [CCST15]	simple	ontology concepts	conj. queries with comparisons
Query refinements			
ConQueR [TC10] TALOS [TCP09, TCP14]	complex	refined query	SPJA SPJ
FlexIQ [ILZ14] Islam *et. al.* [ILZ12]	simple	refined query & Why-Not question refined query	SPJ SPJU
FixTed [BHT15c]	complex	refined query	SPJU
He and Lo [HL14] He and Lo [HL12]	simple	refined query & Why-Not question refined query	top-k (dominating) query top-k query
Zhang *et. al.* [ZHJ$^+$13]	simple	refined query	top-k query
WQRTQ [GLC$^+$15]	simple	refined query & Why-Not question	reverse top-k query
Islam *et. al.* [IZL13]	simple	refined query & Why-Not question	reverse skyline query
Chen *et. al.* [CLJX15]	simple	refined query	spatial keyword top-k query

3.2.1 Instance-Based Explanations

Instance-based explanations consider that the reason for not obtaining the desired results lies with the source data, which are insufficient in their current state to produce the missing answers. Thus, this category proposes the changes need to be made in order to sove the problem.

Non-Answers (NA) [HCDN08] is the first paper to introduce instance-based explanations for missing answers, or -as referenced in the paper - 'provenance tuples' for non-answers. Given a select-project-join (SPJ) query and a missing answer as a simple Why-Not question, the NA proposes to build a *provenance* query, to retrieve the tuples with which the database should be updated. This is a refined statement of the user's query changed based on a number of rules depending on the existing problem setting constraints i.e., the existing data, schema, query and given trustfulness of relations and/or attributes. Moreover the provenance query is executed over a database instance augmented with null-value tuples (null-value having the meaning of variables), in order to be able to represent the tuples missing from the database instance. In this way, the framework proposes how to update the source database, by adding new tuples or modifying existing ones. In order to prune the wide space of possible updates, the framework exploits domain and key constraints. The user can also specify which relations or attributes in the database are trusted and thus are not subject to updates.

Following Non-Answers, Artemis [HH10] computes instance-based explanations for queries containing also aggregation and union (SPJUA) and taking into consideration a complex Why-Not question. An explanation in Artemis is a set of existing tuples and tuples to be inserted in the database, and thus does not consider modification of existing tuples. To compute the set of explanations, Artemis uses the notion of generic witness to capture all the different patterns for the explanations, based on the conditions of the Why-Not question. Then, based on the patterns Artemis transforms the tables in the database to conditional tables [IWL84] containing extra conditional tuples for the missing tuples and executes the query over the new conditional dataset, in an approach resembling the querying over the database enriched with null proxy tuples in NA. Artemis takes into account trusted relations along with some extra techniques, like minimizing the generic witness to narrow down the search space of the explanations. The use of a constraint solver, in combination with unique and key constraints guarantees the correctness of the results. Finally, Artemis provides the opportunity to minimize side effects. Side effects are tuples generated because of the changes in the database, but do not correspond to missing tuples. To minimize the side effects, extra constraints are sent to the constraint solver, built using the values of the side effects and enforcing them to evaluate to false.

Provenance games (PGames) [KLZ13] provide a unified method of answering Why and Why-Not questions by modelling the answer (or the missing-answer) of a query as a won or lost game respectively. In this way, the query is modelled as a game which is instantiated for the (missing) answer, and thus contains the different moves that lead to a win or a loss. The strategy (path) followed to win provides us with the provenance of the answer and the strategy that leads to a loss provides with the instance based explanation (tuples need to exist in the database in order to produce the missing answer). The provenance computed is equivalent to the semiring-provenance[GKT07] for positive relational queries and is additionally able to handle negation in the query. To return a finite set of the explanations, PGames restricts the values in the explanations to values existing in the active domain. To overcome this restriction, and in order to maintain the finiteness of the result [RKL14] extends provenance games to *constraint* provenance games. In the latter, the proposed instance-based explanations are grouped such that they follow the same pattern, satisfying certain common constraints.

Another line of works proposing instance-based explanations, uses the notion of causality paths in order to find and rank explanations. Meliou *et. al.* [MGMS11] theoretically study the unification of instance-based explanations of missing answers and of data present in a query result (a.k.a. their provenance), leveraging the concepts of causality and responsibility. More specifically, given a set of tuples D to be updated in a database (a.k.a. the instance-based explanations) as discussed in NA [HCDN08], a tuple t in D is a cause for a missing answer with a certain amount of responsibility depending on its contingency set. The contigency set of t contains the tuples in D that should be updated along with t so that the missing answer appears in the result. In this way, the responsibility of a cause gives a measure of how important one cause is and is further used to rank the possibly large set of causes. The results apply to conjunctive queries and simple Why-Not questions.

In the same spirit, Roy and Suciu [RS14] use the notion of causality paths in order to rank the obtained explanations. Given the results of several aggregation SQL queries, a user may wonder why there is a specific relationship among the results, for example “Why is the average books price in 2010 higher than in 2013?" (or “Why is the average books price in 2010 not lower than in 2013?"). The explanation here is not directly a set of database tuples like in [MGMS11], but rather a conjunction of predicates over relation *attributes*. In essence, tuples that satisfy this conjunction are responsible for the observed relationship. Since the explanations can be many, the authors propose to compute top-k explanations based mainly on the intervention for each explanation computed based on causality paths over the data built with the aid of foreign keys. The intervention of an explanation measures the degree of influence of this explanation on the query answer and consists in a set of tuples to be removed in order to move towards the desired direction the observed relationship among the query results. To efficiently compute minimal explanations, the method proposes the use of data cubes in database systems that provide this functionality.

Finally, DL-Lite [COSS13] 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).

3.2.2 Query-based explanations

Why-Not [CJ09] is the first paper that explains missing tuples in a query based approach. The followed approach is designed for scientific workflows modelled as Directed Acyclic Graphs (DAG), modelling views (or queries) over relational databases. The nodes of the DAG are workflow manipulations and the leaves are database relations. In the case of relational queries the DAG is a tree with relational operators as nodes. Why-Not takes as input a simple Why-Not question and an SPJU query. Firstly, it identifies data pertinent to the Why-Not question in the input sources. These data, called *unpicked*, should not appear in the lineage [CWW00] of any result tuple. Tracing the unpicked data up the DAG gives rise to *successor* tuples meaning tuples that have in their lineage unpicked data. When at a node of the DAG no more successors of unpicked data are found this node is called a *picky* node and the process stops when no more successors exist on the DAG. The upmost picky nodes are the query based explanations returned by Why-Not.

In this thesis we propose query-based explanations for query debugging, as we consider that the content of the database is trusted and not subject to change, so reasons for the missing-answers can only be placed on the constraints imposed by query operators. But even if the source data were modifiable, existing work [CJ09, Her13] indicate that the numerous instance-based explanations alone are overwhelming and quite costly to compute. Why-Not is the only algorithm producing query-based explanations so we directly compare our proposal, *NedExplain* with Why-not, discussed later in detail in Section 4.1.

Briefly, we argue that Why-Not exhibits a number of shortcomings, when applied to relational queries. The shortcomings are linked to the notion of the unpicked data and successors as defined in the paper which leads to the algorithm failing to produce (correct) explanations in certain cases. Moreover, Why-Not does not behave well in the presence of self-joins and intermediate empty results. These shortcomings result in inaccurate or incomplete explanations, as we also demonstrate experimentally. In addition to dealing with Why-Not shortcomings, with *NedExplain* we also provide a clear formalization of the problem, missing from the Why-Not paper.Moreover, we extend the class of considered queries to unions of aggregate queries, while we provide an algorithm that is competitive with Why-Not in terms of run time.

For completeness, we also report here the second algorithm -*Ted*- proposed in this thesis to compute query-based explanations. The approach followed in *Ted* is completely different than what has been discussed so far, reasoning on the SQL syntax level and not the query tree. Moreover, we propose a novel formalisation of the query-based explanations as a polynomial of query conditions, inspired by the provenance-semirings discussed in the How-Provenance section (Section 3.1). In this ways, we allow for capturing all the ways (query condition combinations) in which a missing tuple was pruned out of the result. We also discuss this approach under different semantics (set, bag, and probabilistic) in Section 4.2. Even though the formal foundation of the *Ted* algorithm is novel, *Ted* is a naive and inefficient implementation of the definitions. For this reason, we propose also the optimized *Ted++* algorithm that renders this approach computationally feasible despite its worst case complexity.

3.2.3 Hybrid explanations

There are cases when a Why-Not question cannot be answered with query-based or instance-based explanations alone. For example, when no compatible data can be computed for the Why-Not question then no query-based explanations can be computed. Moreover, in the presence of (key) constraints there is the probability that the needed tuple insertions are impossible, which are indispensable for instance-based explanations. In such cases, hybrid explanations propose a combined solution with both the updates to be done over the database and the query conditions that still need to be repaired.

In Conseil [Her13, Her15] hybrid explanations are introduced for the first time. The class of queries concerned is relational queries, i.e., select, project, join, union, negation (more specifically one negation is allowed in the proposed algorithm) and aggregation (SPJUAN) queries. The class of Why-Not questions is the one of simple Why-Not questions. Conseil benefits and combines ideas from related works in the query-based and instance-based categories while extending them to non-monotonic queries. Inspired by the query-based approaches based on query trees, the algorithm reasons on a specific query tree representation. First Conseil builds a generic witness, that contains one instance-based and one query-based part. The generic witness is meant to encode the pattern for the hybrid explanations. The instance-part of the generic witness contains conditional tuples satisfying the missing-answers constraints and provide the pattern for compatible tuples. The query-based part contains all the query conditions that can be responsible, that is the selections, joins and negation. At a second step, the compatible tuples are traced through the query tree in a bottom-up way and the parts of the generic witness are assigned with annotations from the set $\{passing, blocking, ambiguous\}$ depending on whether they enable or not (or ambiguous) the compatible tuples to pass to the next node. Note that when compatible tuples are blocked, conditional tuples are generated in their place so as to be able to continue the tracing. Finally, based on the annotated generic witness and a number of derivation rules the hybrid explanations are computed and returned based on a cost function. Since the instance and the query-based explanations co-exist in the generic witness, it is obvious that the query-based explanations are connected to the proposed database updates. As the query-based explanations are computed based on the Why-Not algorithm (in the absence of *NedExplain* at the time), Conseil could transitively be improved using *NedExplain*, as *NedExplain* improves Why-Not.

3.2.4 Ontology-Based Explanations

Recently, the new category of ontology-based explanations have been proposed by Cate *et. al.* [CCST15]. Opposed to the previous categories providing fine-grained explanations (tuples or query conditions) here the explanations are high-level, consisting of concepts from an ontology. The ontology is either provided externally, provided that the database schema can be associated to the concepts of the ontology, or it is derived based on the database schema or database instance at hand. So, a missing-tuple can be associated with a number of concepts. The explanations are intended to provide the *most general* concepts associated with the missing tuple. This means that these concepts are associated with the missing tuples but not with tuples in the query result. The most generak means that the concepts returned as explanations are not subsumed by any other concept being an explanation. The proposed framework considers conjunctive queries with comparisons over a a database schema, an ontology, and a simple Why-Not question. Checking if a most general explanation exists is a NP-complete problem and computing the set of most general explanations is exponential in the size of the database and polynomial time if the database schema is fixed for external ontologies. For ontologies derived from the database the complexity of finding a most general explanation is polynomial for selection-free concept languages [1]. Note that the algorithms as well as the explanations here are completely independent of the query conditions even though the query appears in the framework. Finally, ideas on using hierarchical relationships among categorical attributes have also been proposed in [MK09] for providing refinement-based explanations, further discussed in the next section.

[1] The concept language is used to derive the ontology from the data workspace.

3.2.5 Query Refinements

Once the reasons for unexpected transformation results have been identified, a developer wonders how to leverage this knowledge to obtain the needed results. Often, this requires changing the data transformation (or query in our context), in which case the developer has to manipulate the culprit operators returned as query-based explanations. This task becomes time-consuming when we consider the number of different options of changing a transformation. For instance, given that a particular join is returned as query-based explanation, should she change the join condition, replace it with some outer join, or should it actually be a union?

In this section we review related works proposing how to refine the query or Why-Not question in order to include the specified missing-answers in the result of the refined query. We categorize the related publications based on the kind of Why-Not question (too many/too few results or specific missing tuples) or the type of query ((reverse) top-k, (reverse) skyline, spatial skyline, or traditional relational) they handle. For traditional relational queries, we further identify the approaches that rely on a data classification methodology (classification-based algorithms) and those building query refinements by focusing on and changing the conditions of the query (constrained-based approach).

In this thesis, the proposed query refinements fall into the category of constrained-based relational query refinements. As we discuss in the dedicated chapter (Chapter 5), the main characteristic of our approach is that we leverage the precomputed query-based explanations, obtained for example by the *Ted* algorithm. The majority of the publications presented in this area, do not have the query-based explanations for the Why-Not question as a prerequisite. This is the main difference of our approach in the query refinement explanations field, resulting into the ability to efficiently compute more relevant query refinements.

Too many/too few results Close to the problem of refining a query so as to include specific tuples in the result, is the problem of refining a query so as to meet certain output cardinality constraints, for example when the output of a query contains too many, too few or empty results. Mishra and Koudas [MK09] propose an interactive framework to deal with the too many/too few results problem. The framework involves SPJ queries with disjunctions and conjunctions of range and equality conditions over numerical and categorical attributes. Based on cardinality estimations over samples of the data, the framework decides whether the query should be relaxed (when there are too few results) or constrained (otherwise). The changes on the query *selection* predicates, which consist in removing/adding conjuncts or disjuncts from/in the query's where clause, is performed in a controlled manner based on the user's interaction with the system. For categorical attributes, the hierarchies are either provided by the user or are derived from the database schema. The empty result problem is a subcase of the too few results problem, addressed by Motting *et. al.* in [MMR$^+$13]. In this paper, the authors propose an interactive query relaxation framework based on a probabilistic scheme according to which the user favours the proposed relaxation at each step of the interaction.

An extra requirement set in this work is to maximize a certain objective for example a company's profit. The queries considered here are conjunctive queries with attribute-constant equalities only and the proposed refined queries contain subsets of the initial query conditions that enable some tuples to appear in the result.

The following algorithms consider specific missing answers (unlike the too many/-too few category) from the query result. They are categorized based on the type of query they handle.

Top-k queries He and Lo first addressed the problem of Why-Not questions for *top-k* queries in [HL12], which they extended for top-k dominating queries in [HL14]. A top-k query is a query of the form $q(k, \vec{w})$ that asks for the k best tuples (or points [2]) in a dataset based on a ranking function assigning the weights in $\vec{w}$ to the associated relation attributes (or otherwise called, point dimensions). The Why-Not question consists in a set of missing points M. The answer of the Why-Not question consists in a set of refined top-k queries $q'(k', \vec{w'})$, which include in the k' best results the missing-answers M, however without the restriction that the previous results (or a subset thereof) will also appear in the new result. The user, along with the Why-Not question may provide her preference on whether to favour changes on k or $\vec{w}$ in the refined queries. The authors argue that finding the exact best refinements is a difficult problem and further propose finding the best approximate answers based on a sampling of weighting vectors and optimization techniques that stop or avoid the computation of refined queries that are dominated (i.e., are for sure worse) than others already computed. In a similar approximation approach [HL14] treats also the case of top-k dominating queries, which returns the first k data points depending on the number of data points that they dominate. The Why-Not question again consists in a set of missing data points however the Why-Not answer is a refined query with a new k' and possibly a refined Why-Not question M'. Finally, [ZHJ$^+$13] adopts the techniques discussed in [HL12] to answer another form of Why-Not questions over top-k queries. Here, the Why-Not question consists in two data points, the missing point m and a point p that exists in the top-k results and is compared to m. So, the Why-Not question is of the form *Why is not m in the result since p is in the result?*. The point p is used to add one extra constraint in the problem, by demanding m to be ranked higher than p in the refined query result.

To complement the scenery of Why-Not questions and top-k queries, WQRTQ [GLC$^+$15] addresses Why-Not questions for *reverse* top-k queries, often used for marketing purposes in the enterprise sector. Given a set of tuples (for example products) and a set of weightings (representing customers' preferences on the properties of the products) a reverse top-k query [VDKN10] for a product p returns the customers (i.e., weightings) that contain p in their top-k results. So, a Why-Not question in this context asks why certain customers W are missing from the result of a reverse top-k query [3]. This question can be alternatively stated as why certain customers have

[2] In the paragraphs of top-k and skyline queries, we use the terms tuple and point interchangeably, as tuples are mapped to points on an Euclidean space.

[3] Here, the missing customers are essentially understood as the chosen weightings by the

not the product p in their top-k result. To answer this question, [GLC^{+}15] proposes a unified framework named WQRTQ that computes and provides changes on the product p, and/or the missing-tuples weightings W and k in the optimal way, i.e., minimizing a penalty measuring the difference of the proposed setting w.r.t. the original one.

Recently, a supplementary case of Why-Not questions over top-k queries has been proposed by Chen *et. al.* [CLJX15], not in the relational databases but more Information Retrieval (IR) oriented, focusing on spatial keyword search. The motivation for this work is driven by modern technologies providing many geo-spatial data and making querying them an every day commodity. Every object in the dataset is a pair (location, keyword). A top-k spatial query retrieves the k best objects w.r.t. a query, by best meaning the the closest to the queried location and keyword taking into account also a weighting vector on the two dimensions. A *top-k spatial* why not query asks why a certain object (or set of objects) are not in the top-k returned results. Differently from the previous problems, here the database is not stable but constantly changing and the scoring models are taken from the information retrieval field. Moreover, the algorithms exploit indexing schemes for an efficient time and i/o implementation. The final result is a refined top-k spatial query (i.e., with new k and weighting vector) that includes the missing object(s) in the result.

Skyline queries Except for (reverse) top-k queries, the literature proposes (reverse) skyline queries to express preference over specific tuples and Why-Not questions have been considered in this field as well.

A *skyline* query [BKS01] typically returns these database tuples that have the best values according to a direction (i.e., highest or lowest) on certain attributes (a.k.a. dimensions). Moreover, range constraints can be added along some attributes so that the constrained skyline query [PTFS03] becomes more flexible in describing user's preferences. Chester and Assent [CA15] propose query refinements for Why-Not questions over constrained skyline queries, i.e., questions asking why a certain tuple is not in the skyline returned by the query. Essentially, the algorithm proposed increases the lower limit of the range in the constraint(s) the least possible in order to make the missing tuple appear in the skyline of the new refined query and is based on search space pruning techniques for efficiency. The authors also argue that their method can be trivially extended to return refinements of the missing tuple, which in this case coincide with changes on the source tuple being the missing tuple (i.e., instance-based explanations).

In addition to skyline queries, also *reverse* skyline queries have been considered in the context of Why-Not questions by Islam *et. al.* in [IZL13]. Similarly to reverse top-k queries, reverse skyline queries are also interesting for marketing purposes. The notion of reverse skyline builds on the notion of dynamic skyline [PTFS03] w.r.t. a data point q, which contains the data points that are the closest to the query point q. The reverse skyline [DS07] of a query point q contains the points that have q in their dynamic skylines. A Why-Not question in this case asks why

customers.

a certain point c is missing from the reverse skyline of a query point q. To answer Why-Not questions in this context, [IZL13] proposes refinements of the query point q and/or the missing point c s.t. the refined point c' appears in the reverse skyline of the refined query point q' with the extra requirement that the previous reverse skyline points are not lost. Note that this extra requirement, which also is posed in this thesis proposal for query refinements, is not posed in the rest of the works about top-k, reverse top-k or skyline queries.

Relational queries Now, we visit works on query refinement for relational queries. This is the category that is the most related to our proposal for the query-fixing phase and thus interests us the most. There are two main directions: (i) the classification-based approach and (ii) the constraint-based approach. The first approach builds on the concept of decision trees used for rule derivation and data classification in the machine learning field [Qui87]. The second approach exploits the skyline operator [BKS01] to construct new query conditions.

Classification-based approach The first work we cite following the decision trees approach is TALOS [TCP09, TCP14]. The primary challenge addressed by TALOS is how to generate instance equivalent queries given a result set of tuples T and optionally an initial query Q. Instance equivalent queries generate the same output set T if executed over the same input dataset. This problem can be expressed as a Why-Not query refinement problem, if the output set T consists of the missing tuples and the original query's result. The method considers SPJ queries and complex Why-Not questions, while it is discussed for numerical values only. TALOS builds the possible joins in a refined query Q' based on the input database schema and foreign key constraints. Then, TALOS builds a decision tree beginning with the data corresponding to the result of the joins over the database instance. The decision tree classifies the data on each node into pure (i.e., containing all positive or all negative tuples) and non-pure (i.e., containing both negative and positive tuples) sets of data, using a dynamic labelling scheme. Note that a tuple is labelled as negative if it is not desirable to appear in the query result and positive otherwise.

Each split (branch) in the decision tree, marked on some internal node, is a condition constructed on one attribute and one value. Data satisfying the condition move to the one side and data not satisfying the condition to the other side of the split. The process of splitting t continues until all leaves are pure nodes. The path from the root node to the all positive leaves contain the attribute conditions for the selections of the refined queries. Different branches leading to positive leaves are interpreted as disjunction in the where clause of the query. Moreover, in approximate solutions and in order not to over-constrain the refined query, the at-least one semantics are taken into account. Finally, the refined queries are ranked based on similarity and precision/recall metrics.

The classification approach is also partially followed in [ILZ12], where Islam *et. al.* address in a unified manner unexpected and missing answers, i.e., Why and Why-Not questions. More specifically, given a query containing conjunctions or disjunctions of conditions a user indicates to the system which tuples they expected to be in the result and which they did not. Based on this feedback, the proposed solution treats unexpected tuples with a decision tree approach that classifies result tuples to positives and false positives. The nodes that lead to positive leaves provide the new selections of the refined queries. Note that the classifier used here is a different one than in TALOS [TCP09]. As far as the solution for the missing answers is concerned, the method followed is a constrain-based one. The method begins by identifying common query-based explanations for groups of missing answers and relaxes the conditions in the query-based explanations accordingly. The Why-Not questions addressed are simple, since the missing answers contain attribute-value conditions. Furthermore, only simple query conditions (i.e., selections) are considered for repairing in the refined queries. Finally, the refined queries are ranked based on similarity and precision/recall metrics as in [TCP09] even though the ranking functions used are different.

Constraint-based approach Conquer [TC10] is the first constrained-based approach that we review. Conquer considers in its input a select-project-join-aggregation (SPJA) query and a complex combination of missing tuples. It returns a set of queries that include in their results the missing tuples in addition to the original query result tuples. In the case of SPJ queries, Conquer first searches for refinements including only changes on the selection predicates of the query, and if such changes are not feasible, it searches for refined queries that potentially have a different schema from the original query. In more detail, Conquer begins by computing query refinements that satisfy the requirements, i.e., generate a result that contains the original one and the missing tuples. To do so, Conquer considers the attributes on which the original query poses selection conditions and the values of original result and missing tuples on these attributes. Then, using only the skyline tuples on the dimensions defined by the aforementioned attributes, it computes a set of refined queries, by rewriting the original query conditions with values taken from the skyline tuples. Finally, from the resulting refined queries, only the ones with low *imprecision* are going to be considered for further refinement. The imprecision of a query is measured by the number of irrelevant tuples introduced in the result. Irrelevant are tuples in the refined query result that do not exist in the original query result. The set of queries generated in the first step of the Conquer algorithm, are forwarded to the second step, where they are further refined in order to lower their imprecision metric. In this step, Conquer introduces conditions on attributes from the input query schema, not constrained in the original query. As the problem of minimizing false positive tuples by introducing the minimal number of additional predicates in NP-hard, Conquer uses heuristics to compute how many predicates to add.

When, Conquer is not able to find any refinements based only on selection conditions, it proceeds with a heuristic to compute query refinements with a different $FROM$ clause than the original result. In more detail, Conquer searches in the database schema for combinations of relations that can produce the same output type as the original query Q, and moreover can be joined using foreign key constraints.

Conquer ranks the refined queries produced in the two steps, with respect to the dissimilarity and imprecision metrics. The dissimilarity of a refined query w.r.t. the original is computed based on how many conditions are changed and the type (selection or join) of the changed operator. Finally it creates the skyline of the queries that lower the two metrics, which form the proposed query refinements.

In the same spirit as [ILZ12], FlexIQ [ILZ14] is a user-interactive system targeted to Why and Why-Not questions simultaneously. As such, the user specifies both the expected (but missing) and unexpected (but returned) tuples in the result of a query Q. Here however the queries involved are SPJ queries and the output is not only refined queries but also refined Why-Not questions. FlexIQ considers missing and unexpected tuples together in the algorithmic steps. Mapping the database tuples to points of a space, the algorithm computes the boundary line limiting the query result and tries to move it in the space in the best way so as to exclude the unexpected and include the missing tuples (points) in the result (i.e., below the boundary). The boundary line is a variation of the skyline. As a reminder, a point t dominates a point t' if t is better than t' in at least one dimension and at least as good as t' in all other dimensions. So, whereas the skyline contains all points not dominated by others, the boundary contains all points not dominating others. This explains how boundary sets the limit of the query result and how moving it accordingly can include/exclude tuples. To compute the boundary tuples the skyline operator is taken into account. When the exact solution is not possible due to potential conflicts or redundancies in the user feedback, FlexIQ provides approximate refinements that take into consideration a refined set of missing tuples. The refined queries are disjunctions of conjuncts. In order to create refinements that do not overwhelm the user with their number of conjuncts, FlexIQ combines when possible the conjuncts in the refined queries into one based on a replacement strategy.

In this thesis we propose the *FixTed* algorithm (Chapter 5) to produce refined queries following a constrained-based approach. We argue that this approach is more appropriate for our setting, on the one hand. On the one hand, the classification-based approach is proven, both by Conquer and FlexIQ, to be inefficient especially for big and diverse datasets. On the other hand, starting from the query-based explanations (i.e., the erroneous query conditions) a constrained-based approach appears to be more relevant.

FixTed algorithm is inspired by Conquer. However, Conquer does not use any knowledge of what is wrong with the query, and solely rely on the set of compatible data to produce the query refinements. On the contrary, the *FixTed* algorithm has a different starting point than Conquer, taking advantage of the already computed query-based explanations in the form of polynomials, provided by our *Ted* algorithm (see Section 4.2). In this way, *FixTed* guarantees to produce the lowest dissimilarity query refinements, by adopting a more sophisticated cost model too. Briefly, the dissimilarity is captured by the value difference or edit distance of changed conditions and the number of changed or added conditions. Moreover, *EFQ* computes refinements more efficiently by being able to reduce the dimensions of the skyline tuple computation. Finally, *EFQ* is aware of the type of conditions that should be changed (a.k.a., if joins should be changed as well) from the beginning. Thus, it produces targeted query refinements depending on the type of the conditions involved in the query-based explanations. As we discuss in Chapter 5, *EFQ* proposes a different class of refined queries, i.e., left/right outer-join queries, when joins are involved in the query-based explanations.

3.3 Summary

In this section, we reviewed some of the most important publications on the subjects that this thesis addresses. In a first level we distinguish between works investigating why certain data exist in the output of a data transformation or more specifically in a query result and works investigating why there are not some other expected data in the result. The first category is also widely known as the data provenance problem, linking output data with their origins in the source data. Seminal publications in this category include lineage ([CW01]) and provenance semirings ([GKT07]), concepts also used in this thesis' proposed framework.

This thesis falls in the second category, that is the Why-Not provenance category. In this field of research, different algorithms have been proposed to answer Why-Not questions either by debugging the input query (like [CJ09]) and/or database (like [HCDN08, HH10, Her15, CCST15]), or by proposing alternative query refinements that include the missing answer in their results (like [TC10]). Furthermore, we placed this thesis algorithms in the sub-categories of query-based explanations (*NedExplain* and *Ted*) and refinement-based explanations (*FixTed*) and highlighted the most important differences and competences of our approaches w.r.t. the state of the art algorithms. More theory and technical details necessary for the exact comparison are provided in the dedicated chapters.

Chapter 4

Query Debugging

In this chapter we describe *NedExplain* and *Ted*, the two proposals developed in this thesis to address the query debugging problem, as introduced in Problem Statement 2.3.1. The objective of the two algorithms is the same as they both solve an explanation scenario by providing query-based explanations. Thus, the general notions and definitions described in Chapter 2 are used for both algorithms. However, there are subtle differences w.r.t. their application scope and principles. For example, each algorithm addresses a different class of queries and Why-Not questions. Moreover, the output Why-Not answer granularity and format differs so each algorithm complies with its own specific Why-Not answer definition.

The chapter is organised as follows. Section 4.1 is dedicated to *NedExplain*, and provides its general contribution, and the description and algorithmic steps to produce the Why-Not answer. Moreover, we provide the experimental evaluation of *NedExplain*, including a comparative evaluation to the state of the art algorithm *Why-Not* [CJ09] in terms of run time efficiency and answer quality. Section 4.2 describes *Ted* in a similar structure. In the beginning the contributions are provided followed by the *Ted* and the efficient *Ted++* algorithms producing the Why-Not answer in this context. Then, an experimental evaluation follows comparing *Ted* and *Ted++* with the state of the art algorithms, again both in terms of run time performance and Why-Not answer quality. Finally, we provide a theoretical discussion extending the *Ted* Why-Not answer to different database and query classes.

Publications NedExplain has been published as a full paper [BHT14c] in the proceeding of the *International Conference on Extending Database Technology (EDBT) 2014*. It was also presented ([BHT13]) in the french conference on databases *Bases de Données Avancées (BDA) 2013*.

A preliminary version of the Why-Not answer polynomials, along with the naive algorithm *Ted* were published as a workshop paper [BHT14a] at the *International Workshop on Theory and Practice of Provenance (TaPP) 2014*. This version [BHT14b] was also presented in the French database conference on databases *Bases de Données Avancées (BDA) 2014*. An extended version [BHT15d] including the theoretical discussion on query equivalence, is published as an invited article in the *Ingénierie* *des Systèmes d'Information (ISI) journal 2015.* The optimized algorithm *Ted++* is published as a full paper ([BHT15a]) in the proceedings of the *International Conference on Information and Knowledge Management (CIKM) 2015* and ([BHT15b]) in *Bases de Données Avancées (BDA) 2015.*

Finally, the first results on query debugging and the perspectives for using Why-Not answer polynomials for query fixing [Tzo14] were presented at the *Very Large Databases (VLDB) PhD Workshop 2014.*

4.1 NedExplain

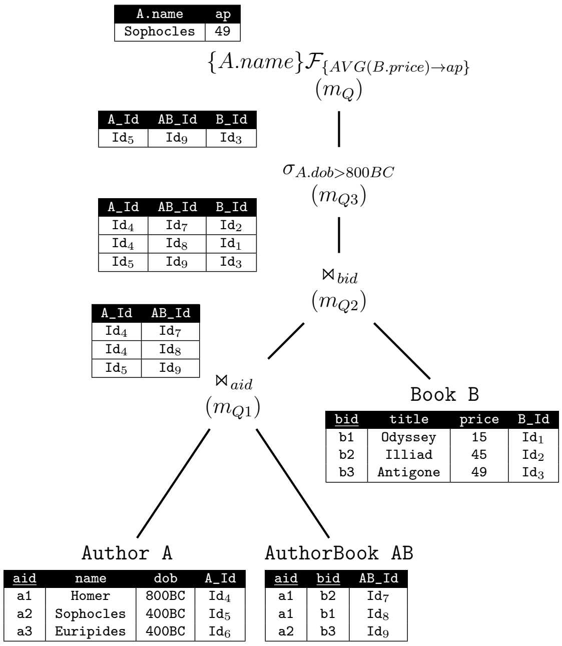
NedExplain [BHT14c] is an algorithm that computes query-based explanations given an explanation scenario Δ (see Section 2.3). More specifically *NedExplain* considers Select-Project-Join-Aggregate queries and unions thereof (SPJUA) and simple Why-Not questions. We motivate the problem and the proposed solution with the following example that is also used as the running example for this section.

```
SELECT A.name, AVG(B.price) AS ap
FROM Author A, AuthorBook AB, Book B
WHERE A.dob > 800BC
    AND A.aid = AB.aid
    AND B.bid = AB.bid
GROUP BY A.name
```

(a) SQL query *Q*

```
{A.name = Homer, ap > 25}
```

(b) Why-Not question *WN*



A.name	ap
Sophocles	49

A_Id	AB_Id	B_Id
Id_5	Id_9	Id_3

A_Id	AB_Id	B_Id
Id_4	Id_7	Id_2
Id_4	Id_8	Id_1
Id_5	Id_9	Id_3

A_Id	AB_Id
Id_4	Id_7
Id_4	Id_8
Id_5	Id_9

Book B

bid	title	price	B_Id
b1	Odyssey	15	Id_1
b2	Illiad	45	Id_2
b3	Antigone	49	Id_3

Author A

aid	name	dob	A_Id
a1	Homer	800BC	Id_4
a2	Sophocles	400BC	Id_5
a3	Euripides	400BC	Id_6

AuthorBook AB

aid	bid	AB_Id
a1	b2	Id_7
a1	b1	Id_8
a2	b3	Id_9

(c) Query tree

Author

aid	name	dob	A_Id
a1	Homer	800BC	Id_4
a2	Sophocles	400BC	Id_5
a3	Euripides	400BC	Id_6

AuthorBook

aid	bid	AB_Id
a1	b2	Id_7
a1	b1	Id_8
a2	b3	Id_9

Book

bid	title	price	B_Id
b1	Odyssey	15	Id_1
b2	Illiad	45	Id_2
b3	Antigone	49	Id_3

(d) Database instance $\mathcal{I}$ over $\mathcal{S}$=$\{Author, Book, AuthorBook\}$

Figure 4.1: Scenario for *NedExplain* running example

Example 4.1.1. *Consider a database $\mathcal{S}$ with the relations* `Author (A)`, `Book (B)` *and* `AuthorBook (AB)`. *A developer executes a query over this database to find ancient authors and the average price of their books. This query is expressed as the SQL query shown in Figure 4.1(a), while a tree representation can be seen in Figure 4.1(c) - ignore the operator labels m_{Qi} in the query tree for now. Let us further consider the database instance $\mathcal{I}$ over $\mathcal{S}$ shown in Figure 4.1(d). Based on the data and query, the query result $Q[\mathcal{I}]$ contains only one tuple:* (*Sophocles, 49*).

Having retrieved only one tuple, the developer wonders why there is no tuple with author name Homer and average price greater than 25 in the query result, as he expected. This Why-Not question is expressed by WN in Figure 4.1(b). To answer this Why-Not question we can see that the selection on the attribute `dob` *is too strict to let the author named Homer pass. Indeed the* compatible *source tuple Id_4=(a1, Homer, 800BC), is pruned out from the result, leaving no* successor *tuples of Id_4 in the output of the selection. Note that here and in what follows, we may refer to a tuple from a relation R using the R_Id attribute.*

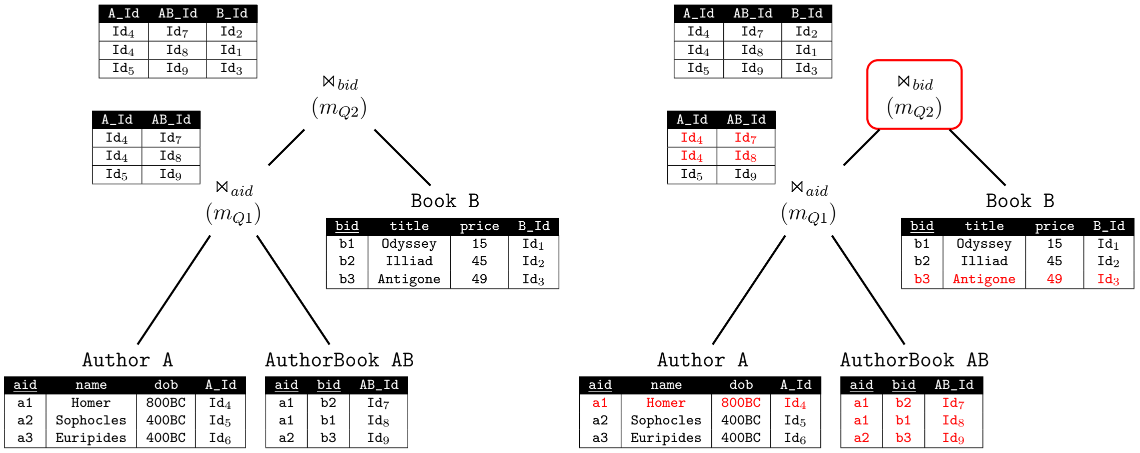
Now, let us change the Why-Not question to ask 'Why are there no result tuples with a name different from Sophocles?' One explanation for not obtaining any other name than Sophocles, is again the selection $\sigma_{A.dob>800}$, for pruning out of the result the compatible tuple Id_4 with the name Homer. Moreover, the join $\bowtie_{aid}$ prunes out the compatible tuple Id_6=(a3, Euripides, 400BC) that could contribute to a result tuple with the author name Euripides. So, for this Why-Not question two query-based explanations exist in the form of picky *operators (an operator is picky when it prunes out compatible tuples): the selection $\sigma_{A.dob>800BC}$ and the join $\bowtie_{aid}$.*

The state-of-the-art algorithm computing query-based explanations, called *Why-Not* algorithm [CJ09], is designed for workflows but also applies to relational queries when considering relational operators as the individual manipulations of the workflow, as discussed in Section 3.2. However, the *Why-Not* algorithm makes use of two central definitions that may yield incomplete or even incorrect results. For this reason, now we stress out the cases when the *Why-Not* algorithm does not return correct or complete query-based explanations in contrast with our proposal, by emphasising how the relevant definitions in the two frameworks differ.

4.1.1 *NedExplain* Algorithm vs *Why-Not* Algorithm

Overall, the shortcomings of [CJ09] are linked to processing queries with self-join or empty intermediate results, or are linked to the formulation of insufficiently precise Why-Not answers or the incapability of providing a Why-Not answer at all. These come as a result of the inappropriate definition of compatible source data and their successors as well as the definition of a query-based explanation in [CJ09]. To describe each case, we use variations of the example in Figure 4.1, which we run using both the *NedExplain* and *Why-Not* algorithms, highlighting the differences between the two. Note that for convenience, we refer to tuples generated in intermediate nodes, using the identifier attributes $\_Id$, however it should be understood that all intermediate tuples come with all their attributes.

***Why-Not* algorithm does not compute any explanation** Consider the subquery Q_2 of our running example in Figure 4.1(c). This query returns all the authors and their books in our database and projects out all the involved attributes. Next, note that we refer to tuples resulting from joins over source tuples, by the concatenation of the involved source tuple identifier attributes $\_Id$. Figure 4.2(a) shows the data-flow on the query tree using the *Why-Not* algorithm and Figure 4.2(b) the dataflow following *NedExplain*. The output of Q_2 consists of three tuples: $\{Id_4Id_7Id_2,\ Id_4Id_8Id_1,\ Id_5Id_9Id_3\}$. Let us now consider the Why-Not question $\{A.name{=}Homer, B.price{=}49\}$: Why does not the output of Q_2 contain any tuple of *Homer*, with a book price *49*?



A_Id	AB_Id	B_Id
Id_4	Id_7	Id_2
Id_4	Id_8	Id_1
Id_5	Id_9	Id_3

A_Id	AB_Id
Id_4	Id_7
Id_4	Id_8
Id_5	Id_9

bid	title	price	B_Id
b1	Odyssey	15	Id_1
b2	Illiad	45	Id_2
b3	Antigone	49	Id_3

aid	name	dob	A_Id
a1	Homer	800BC	Id_4
a2	Sophocles	400BC	Id_5
a3	Euripides	400BC	Id_6

aid	bid	AB_Id
a1	b2	Id_7
a1	b1	Id_8
a2	b3	Id_9

(a) *Why-Not*

A_Id	AB_Id	B_Id
Id_4	Id_7	Id_2
Id_4	Id_8	Id_1
Id_5	Id_9	Id_3

A_Id	AB_Id
Id_4	Id_7
Id_4	Id_8
Id_5	Id_9

bid	title	price	B_Id
b1	Odyssey	15	Id_1
b2	Illiad	45	Id_2
b3	Antigone	49	Id_3

aid	name	dob	A_Id
a1	Homer	800BC	Id_4
a2	Sophocles	400BC	Id_5
a3	Euripides	400BC	Id_6

aid	bid	AB_Id
a1	b2	Id_7
a1	b1	Id_8
a2	b3	Id_9

(b)*NedExplain*

Figure 4.2: (a) *Why-Not*, and (b) *NedExplain* algorithms for the case when the *Why-Not* algorithm does not compute a Why-Not answer.

To answer this Why-Not question both algorithms start by identifying the source data that are relevant w.r.t. the Why-Not question, i.e., the compatible tuples, and which are highlighted on the leaves of the two trees. We also highlight successor tuples, i.e., intermediate result tuples that are associated with some compatible tuple. We can see that in Figure 4.2(a) there are no highlighted tuples on the leaves. This means that the *Why-Not* algorithm does not identify any compatible tuples.

The reason behind this is the definition of compatible tuples of [CJ09]: a compatible tuple is a tuple

— that belongs to a source relation,

— satisfies the conditions in the Why-Not question posed over the attributes of this relation, and

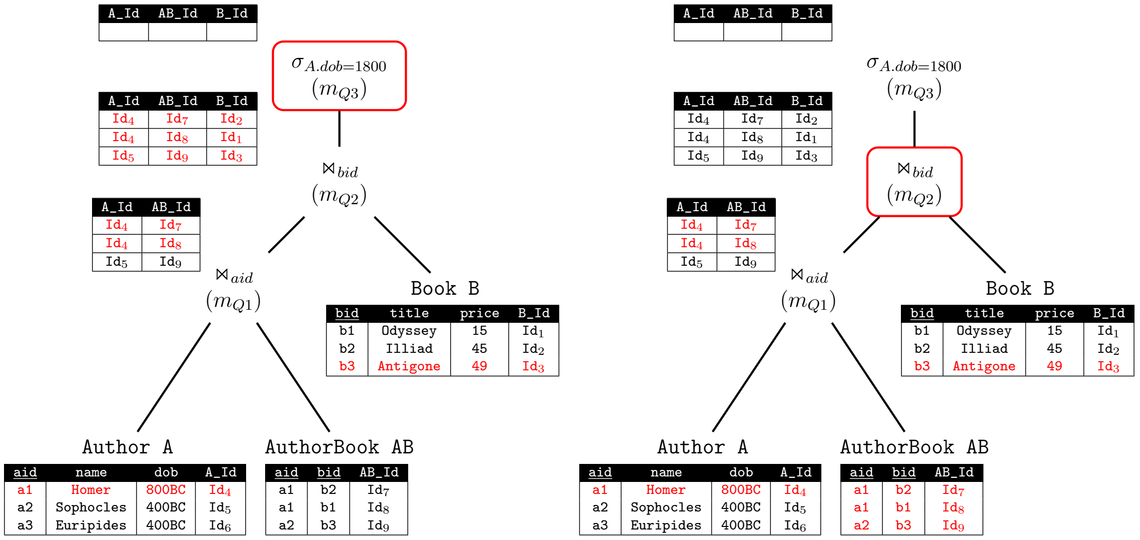
— is not contained in the lineage (see Definition 3.1.1) of any query result tuple.

In our example the tuple Id_4 in `Author` has the name `Homer` so satisfies the condition $A.name{=}Homer$. However, the tuple Id_4 is in the lineage of the query result tuples $Id_4Id_7Id_2$ and $Id_4Id_8Id_1$, so Id_4 is not a compatible tuple according to the definition in [CJ09]. In the same way, the tuple Id_3 from the Book relation is not compatible because it is in the lineage of the result tuple $Id_5Id_9Id_3$.

As a consequence, the *Why-Not* algorithm cannot compute any query-based explanation (i.e., Why-Not answer).

NedExplain on the contrary correctly identifies the two tuples Id_4 and Id_3 as compatible tuples, as the constraint about lineage is not part of our definition. Then, the two compatible tuples are traced up the tree until we lose their trace in the output of m_{Q_2}. So, m_{Q_2} is correctly identified as a query-based explanation by *NedExplain*. Note also that all tuples in AuthorBook are considered also compatible w.r.t. the Why-Not question by *NedExplain*. Even though the Why-Not question does not specify conditions over the attributes of these tuples, still (some of) these tuples are indispensable for the production of compatible tuples successors. Their use is shown shortly after, in the case of empty intermediate results.

***Why-Not* algorithm computes inaccurate explanations** To illustrate this, let us change the subquery Q_3 of our running example to $\sigma_{A.dob=1800}$ and consider again the Why-Not question $\{A.name{=}Homer, B.price{=}49\}$ on the output of Q_3, which is now empty. Figure 4.3(a) shows the data-flow on the query tree for the *Why-Not* algorithm and Figure 4.3(b) the dataflow for *NedExplain*.



A_Id	AB_Id	B_Id

A_Id	AB_Id	B_Id
Id$_4$	Id$_7$	Id$_2$
Id$_4$	Id$_8$	Id$_1$
Id$_5$	Id$_9$	Id$_3$

A_Id	AB_Id
Id$_4$	Id$_7$
Id$_4$	Id$_8$
Id$_5$	Id$_9$

Book B

bid	title	price	B_Id
b1	Odyssey	15	Id$_1$
b2	Illiad	45	Id$_2$
b3	Antigone	49	Id$_3$

Author A

aid	name	dob	A_Id
a1	Homer	800BC	Id$_4$
a2	Sophocles	400BC	Id$_5$
a3	Euripides	400BC	Id$_6$

AuthorBook AB

aid	bid	AB_Id
a1	b2	Id$_7$
a1	b1	Id$_8$
a2	b3	Id$_9$

(a) *Why-Not*

A_Id	AB_Id	B_Id

A_Id	AB_Id	B_Id
Id$_4$	Id$_7$	Id$_2$
Id$_4$	Id$_8$	Id$_1$
Id$_5$	Id$_9$	Id$_3$

A_Id	AB_Id
Id$_4$	Id$_7$
Id$_4$	Id$_8$
Id$_5$	Id$_9$

Book B

bid	title	price	B_Id
b1	Odyssey	15	Id$_1$
b2	Illiad	45	Id$_2$
b3	Antigone	49	Id$_3$

Author A

aid	name	dob	A_Id
a1	Homer	800BC	Id$_4$
a2	Sophocles	400BC	Id$_5$
a3	Euripides	400BC	Id$_6$

AuthorBook AB

aid	bid	AB_Id
a1	b2	Id$_7$
a1	b1	Id$_8$
a2	b3	Id$_9$

(b)*NedExplain*

Figure 4.3: (a) *Why-Not*, and (b) *NedExplain* algorithms for the case when the *Why-Not* algorithm computes an inaccurate explanation.

To answer this Why-Not question, the *Why-Not* algorithm identifies as compatible the tuple Id_4 from the Authors relation and the tuple Id_3 from the Books relation (see Figure 4.3(a) the highlighted tuples in the leaves). Now, this has been possible because these two tuples are not in the lineage of any result tuple, since the query result set is empty. *NedExplain* also identifies the same compatible tuples in relations `Author` and `Book` and also all the tuples in AuthorBook, as seen in the bottom of Figure 4.3(b).

Then, the *Why-Not* algorithm traces in the tree the compatible tuples by identifying their successors in each node (a.k.a. query operator). The successors are highlighted in the output of each node. In [CJ09] a tuple from the output of a subquery is defined as a successor of a compatible tuple if the compatible tuple is contained in the lineage of this output tuple. Following this definition we can see that we still can find successors of the compatible tuples until m_{Q_3}, where the trace of both compatible tuples is lost. So, the Why-Not answer returned by the *Why-Not* algorithm is m_{Q_3}. However, as shown before, Homer is not associated to a book with price 49 which means that the join in Q_2 is also responsible for not outputting the desired result, a fact not reflected in the answer returned by *Why-Not*.

NedExplain introduces a less permissive notion of successor tuples, allowing only compatible tuples to exist in the lineage of the successors. In this way, the tuples $Id_4Id_7Id_2$, $Id_4Id_8Id_1$, and $Id_5Id_9Id_3$ in the output of m_{Q_2} are not successors w.r.t. *NedExplain*. Indeed none of these tuples contain both Id_4 and Id_3 in the same tuple. In general, at least one no-compatible tuple id (like Id_1, Id_2 or Id_5) participates in each output tuple of m_{Q_2}, a condition that suffices for not considering any of them as *valid* successors.

So, m_{Q_2} is identified as a query-based explanation and is included in the Why-Not answer, an explanation missed by *NedExplain*.

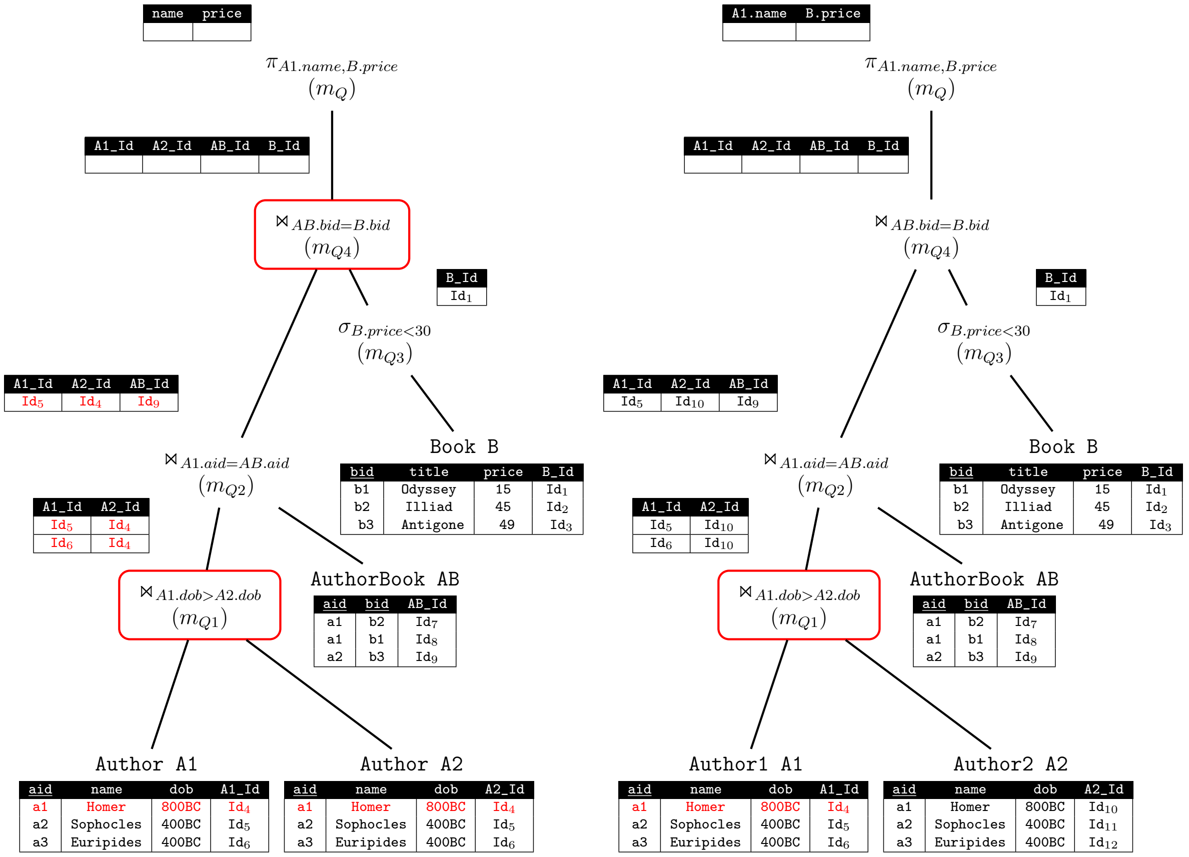
***Why-Not* algorithm does not treat self-join correctly** To illustrate this case, we build a new query involving a self join which is depicted in Figure 4.4. This query asks for authors that were born more recently than others and whose books price is lower than 30. The result of the query is empty. We wonder why Homer is not in the result set. In this case, the Why-Not question for the two algorithms is different:

— $\{name{=}Homer\}$ for *Why-Not*
— $\{A1.name{=}Homer\}$ for *NedExplain*

Note that *Why-Not* and *NedExplain* did not express the Why-Not questions in the same way, neither in the previous examples. However, since for the previous examples this did not pose a problem, and to simplify the examples, we considered that their formatting was the same for the Why-Not question. Here, we show how not considering qualified attributes in the Why-Not question may yield inaccurate results.

Again the first step towards answering the Why-Not question, is to identify the compatible tuples. As defined in [CJ09], a compatible tuple belongs to a relation that has an attribute in the Why-Not question: in this case the attribute *name*. So, the compatible tuples are looked for in both instances of `Author` as seen in Figure 4.4(a). This is a poor decision however, because only tuples with the name *Homer* from the first instance can produce the missing tuple, because this is the projected attributed by the query.

So, considering compatible tuples from both `Author` instances yields two explanations: the join on m_{Q_1} and the join on m_{Q_4}. Indeed, m_{Q_1} does not output



(a) *Why-Not* (b) *NedExplain*

Figure 4.4: (a) *Why-Not*, and (b) *NedExplain* algorithms for the case of self-join.

any successor for the compatible tuple Id_4 coming from the instance A_1, so m_{Q_1} is picky for this compatible tuple. The second join does not output any successor for the *false-identified* compatible tuple Id_4 coming from the instance A_2. Thus, m_{Q_2} is picky for this tuple and is considered as picky 'by mistake'. It is the author *Sophocles* that is excluded from the result of m_{Q_2} and not *Homer*.

Thus, in the case of self joins, or other cases when the same attribute is used in different relations, the *Why-Not* algorithm may compute misleading explanations.

On the other side, in *NedExplain* we use qualified attributes and so no attribute can appear in more than one relation. So, when dealing with self-join, the two participating instances have a different name, and a different schema, although the same content (except for *Id*s). So, in this case, *NedExplain* identifies only on compatible tuple, the tuple $A1\_Id{:}Id_4$ from the relation $A1$, as seen in Figure 4.4(b). Thus, it correctly computes only the join m_{Q_1} as an explanation.

Note that in Figure 4.4 the attribute `name` of `Author1` is referenced in *NedExplain* as `Author1.name`, and the same holds for all the attributes.

4.1.2 Contribution

The previous observations w.r.t *Why-Not* [CJ09] have motivated us to investigate a novel algorithm, named *NedExplain*[1]. Our contribution is:

Formalization of query-based Why-Not provenance We provide a formalization of query-based explanations for Why-Not questions that was missing in [CJ09]. It relies on new notions of compatible tuples and of their valid successors. This definition subsumes the concepts informally introduced previously. It covers cases that were not properly captured in [CJ09]. Moreover it takes into account queries involving aggregation (i.e., select-project-join-aggregate queries, or SPJA queries for short) and unions thereof.

The *NedExplain* Algorithm Based on the problem formalization, the *NedExplain* algorithm is designed to correctly compute query-based explanations given an explanation scenario over the class of unions of SPJA queries and a Why-Not question as in Definition 2.2.2.

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

Detailed analysis of *Why-Not* We review in detail *Why-Not* [CJ09] in the context of positive relational queries and show that it has several shortcomings leading it to return no, partial, or misleading explanations.

4.1.3 Preliminaries

In this section we revisit the notions introduced in Chapter 2 and contextualize them for the purpose of the NedExplain algorithm.

The main characteristic of NedExplain is that it uses a query tree representation of the input query to trace the compatible data and to finally produce the Why-Not answer. More specifically, *NedExplain* considers an SPJUA query Q as in Definition 2.1.9, involving attribute renamings happening through *joins* and *unions*. So, it is possible that renamed attributes are projected out in the output schema of the query Q.

Being specified over the output schema of Q, the Why-Not question can be specified over such renamed attributes as well. However, in order to compute the compatible tuples that reside in the input query instance, it is essential that the

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

Why-Not question is specified over attributes of the input schema of the query. This is clear if we recall that the compatible tuples are the result of the evaluation against $\mathcal{I}_Q$ of the query Q_{WN} (Definition 2.2.5) corresponding to the Why-Not question.

For example, consider a new scenario, where the considered query is query Q_2 rooted at m_{Q_2} (ignore the higher nodes) on the tree of Figure 4.1(c), page 44. The output schema of Q_2 contains the renamed attribute *bid* introduced by the renaming $(B.bid, AB.bid, bid)$ linked to the join operator $\bowtie_{bid}$. In this new sceanrio, the user can specify a Why-Not question over the renamed attribute. However, in order to be able to compute the compatible tuples in the source relations, *bid* should be resolved to the source relation attributes involved in the renaming, that is $B.bid$ and $AB.bid$. For this reason, we map the Why-Not question to attributes that appear in the input schema, using the definition of *unrenamed* Why-Not question as follows.

Definition 4.1.1 (Unrenamed Why-Not question w.r.t. a query Q)**.** *Let WN be a Why-Not question. Given a renaming ν as in Definition 2.1.5, $\nu_{|1}^{-1}(WN)$ (respectively $\nu_{|2}^{-1}(WN)$) is obtained from WN by replacing A_{new} in WN by A_1 (respectively A_2) for each $(A_1, A_2, A_{new}) \in \nu$. Now, let Q be a query. The mapping UnR_Q associates to WN a Why-Not question or a general Why-Not question defined by:*

1. *if $Q = [R_i]$ then $UnR_Q(WN) = WN$,*
2. *Let Q_1, Q_2 be queries*
   (a) *if $Q = [Q_1] \bowtie_\nu [Q_2]$, then* $UnR_Q(WN) = UnR_{Q_1}(\nu_{|1}^{-1}(WN)) \cup UnR_{Q_2}(\nu_{|2}^{-1}(WN))$
   (b) *if $Q = [Q_1] \cup_\nu [Q_2]$, then* $UnR_Q(WN) = \{UnR_{Q_1}(\nu_{|1}^{-1}(WN))\} \cup \{UnR_{Q_2}(\nu_{|2}^{-1}(WN))\}$
3. *if $Q = \pi_W[Q_1]$, or $Q = \alpha_{G,F}(Q_1)$, or $Q = \sigma_C[Q_1]$ then $UnR_Q(WN) = UnR_{Q_1}(WN)$.*

Intuitively the previous definition distinguishes the unrenaming process in the case of attributes in the Why-Not question introduced by a join-renaming or by a union-renaming. In the case of join-renamed attribute (Definition 4.1.1-2a), we replace each condition involving this attribute by two conditions with the associated source relation attributes. For example, consider the relations R and S with the join-renaming $(R.A, S.A, A')$ and the Why-Not question WN=$\{A'$= 4$\}$. Then, the unrenamed Why-Not question is $UnR_Q(WN)$=$\{R.A$= 4, $S.A$= 4$\}$. In the case of union-renamed attributes (Definition 4.1.1-2b) we create one duplicate of the Why-Not question for each associated source relation attribute. The resulting Why-Not question is the union of these Why-Not questions. For the previous example, if $(R.A, S.A, A')$ is a union-renaming, the result of the unrenaming of WN=$\{A'$= 4$\}$ is $\{\{R.A$= 4$\}, \{S.A$= 4$\}\}$.

Example 4.1.2. *Consider again the example query Q and data in Figure 4.1 and assume that Q outputs one more attribute, i.e., Γ_Q=$\{A.name, aid, ap\}$. Consider also the renaming ν=$\{(AB.aid, A.aid, aid)\}$. For the Why-Not question WN=$\{A.name$ =*

$Homer, aid = a1, ap = x_1\}$, the attribute `aid` can be unrenamed to A.aid and to AB.aid, two qualified attributes that cannot be further unrenamed. So, the unrenamed predicate WN is $\{A.name = Homer, A.aid = a1, AB.aid = a1, ap = x_1\}$. Note that ap is a new attribute introduced by an aggregation function, so it does not take part in the unrenaming of WN.

For the rest of the discussion when referring to *WN* we mean its unrenamed version. Having this, we can compute the compatible tuples w.r.t. *WN* from the source database instance, in the already discussed manner. This is also the first step towards answering the Why-Not question.

To model this first step on the query tree, we identify the compatible tuples on the tree leaves. The compatible tuples stored in each leaf are *partial* compatible tuples, originating from the relation associated to the leaf. It is then clear, that *NedExplain* is tailored only for simple Why-Not questions, as it can accommodate only intra-relation conditions. For a reminder, if CT is the set of compatible tuples, then the set of partial compatible tuples $CT_{|R}$ from a relation R is obtained by evaluating the query $(R, \mathcal{A}(R), WN_{|R})$ over $\mathcal{I}_{|R}$. In the rest of this section on *NedExplain*, compatible tuples designate partial compatible tuples unless otherwise stated.

Example 4.1.3. *The Why-Not question of our running example is split into WN_{conj}=$\{A.name$=$Homer\}$ and WN_{α}=$\{ap$>$25\}$. Only WN_{conj} is used for computing the compatible tuples. The condition A.name=Homer is over the relation A and leads to the compatible tuple $Id_4 \in \mathcal{I}_{|A}$ (see Figure 4.1(b)). All the tuples in the relations B and AB are compatible, since no conditions are posed over attributes in B and AB.*

A Why-Not question *WN* may be specified over attributes originating from a subset of the input schema $\mathcal{S}_Q$. So, only some of the input schema relations are constrained by the conditions in *WN* and more specifically by the conditions in its conjunctive part WN_{conj}. Let $\mathcal{S}_W$ be the set of relations over which WN_{conj} is defined. To distinguish between compatible tuples that originate from $\mathcal{S}_W$ and those originating from $\mathcal{S}_Q \setminus \mathcal{S}_W$, we define the *direct* and *indirect* compatible tuple sets. The direct compatible tuple set *Dir* equals to

$$Dir = \bigcup_{R \in \mathcal{S}_W} CT_{|R}$$

The indirect compatible tuple set *InDir* equals to

$$InDir = \bigcup_{R \in \mathcal{S}_Q \setminus \mathcal{S}_W} CT_{|R}$$

By definition it holds that one direct tuple cannot be indirect and vice versa, thus $Dir \cap InDir$=$\emptyset$.

The distinction of compatible tuples to direct and indirect compatible tuples plays a central role in the computation of the Why-Not answer as shown in the next section.

Example 4.1.4. *Pursuing Example 4.1.3, Dir=$\{Id_4\}$ whereas InDir=$\mathcal{I}_{|AB} \cup \mathcal{I}_{|B}$.*

4.1.4 Why-Not Answer

To answer the Why-Not question we need to find the nodes of the query tree where we lose compatible tuples. For this reason, we trace the compatible tuples (direct and indirect) in a bottom up way on the query tree, computing along the way the nodes where the trace of some compatible tuple is lost. In other words this means that we identify the subqueries of Q that do not produce successors (formally defined below) of some compatible tuple.

In order to do this, we associate an operator m_{Q_i} to each subquery Q_i that serves as its type signature. For instance in Figure 4.1, the operator (m_{Q1}) associated with the subquery Q_1 is $A \bowtie_{aid} AB$. The input instance $\mathcal{I}_i$ of an operator m_{Q_i} includes solely the outputs of its direct children in the tree (or, in case of leaf nodes, the instance of the corresponding table). For example, in Figure 4.1 the input instance for m_{Q_2} consists of the output instance of m_{Q_1} and $\mathcal{I}_B$. The output of an operator m over its input instance $\mathcal{I}_m$ is denoted by $m(\mathcal{I}_m)$.

Data lineage, or lineage for short as defined in [CW00] (see also Definition 3.1.1), is at the basis of tuples tracing. As already said, the lineage of a tuple t that appears in the ouput of a query Q (represented as a tree of relational operators) consists of the different sets of input database tuples that participate in the production of the tuple t provided v.

The purpose of the next example is to give the intuition of how lineage is defined for operators and also to explain our notation. Consider two relation schemas $R(A, B)$ and $S(A, B)$ and the database instance $\mathcal{I}{=}\mathcal{I}_R \cup \mathcal{I}_S$, for which $\mathcal{I}_R{=}\{(a_1, b_1), (a_1, b_2), (a_2, b_1)\}$ and $\mathcal{I}_S{=}\{(a_1, b_1), (a_2, b_2)\}$. Consider also the operator $m{=}[R] \cup [S]$, whose evaluation against $\mathcal{I}$ produces

$$m(\mathcal{I}){=}\{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2)\}$$

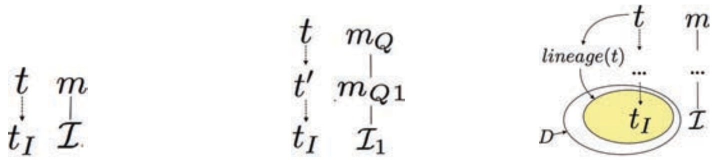
First note that the lineage of t is only defined when $t \in m(\mathcal{I})$. So, the lineage of the tuple $t{=}(a_1, b_1)$ w.r.t. m and $\mathcal{I}$, is defined in [CW00] as a tuple of instances $lineage(t){=}(\mathcal{J}_R, \mathcal{J}_S)$, where $\mathcal{J}_R{=}\{(a_1, b_1)\}$ is an instance over R and $\mathcal{J}_S{=}\{(a_1, b_1)\}$ an instance over S. In our setting, it suffices to accept that the lineage of the tuple t is the union of the subsets $\mathcal{J}_R, \mathcal{J}_S$, i.e., it is the set of tuples (typed tuples)

$$lineage((a_1, b_1)){=}\{(R.A : a_1, R.B : b_1), (S.A : a_1, S.B : b_1)\}$$

Given an operator m and an input instance $\mathcal{I}$, we define that $t{\in}m(\mathcal{I})$ *is a successor of* some $t_{\mathcal{I}}{\in}\mathcal{I}$ by $t_{\mathcal{I}}$ is in the lineage of t w.r.t. m. Figure 4.5(a) illustrates the successor relationship between t and $t_{\mathcal{I}}$ belonging to $m(\mathcal{I})$ and $\mathcal{I}$, respectively.

We now define the notion of tuple successor w.r.t. to a (composed) query. Note that in the following definition, UOp is a unary operator among σ, π, α, and BOp is a binary operator among $\cup$, $\bowtie$. The definition is illustrated in Figure 4.5(b) for the case of unary operators.

Definition 4.1.2 (tuple successor w.r.t. a query)**.** *Let Q be a query over $\mathcal{S}_Q$ and $\mathcal{I}$ be an instance over $\mathcal{S}_Q$. A tuple $t{\in}Q(\mathcal{I})$ is a successor of some $t_{\mathcal{I}}{\in}\mathcal{I}$ w.r.t. Q if*



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

Figure 4.5: Successor t of a tuple $t_\mathcal{I}$

— *for* $Q = UOp[Q_1]$
— *there exists some* $t' \in Q_1(\mathcal{I}_1)$ *s.t.* t *is a successor of* t' *w.r.t.* m_Q *and* $Q_1(\mathcal{I}_1)$ *and*
— $t' = t_\mathcal{I}$ *or*
— t' *is a successor of* $t_\mathcal{I}$ *w.r.t.* Q_1
— *for* $Q = [Q_1]BOp[Q_2]$
— *there exists some* $t' \in Q_1(\mathcal{I}_1) \cup Q_2(\mathcal{I}_2)$ *s.t.* t *is a successor of* t' *w.r.t.* m_Q *and* $Q_1(\mathcal{I}_1) \cup Q_2(\mathcal{I}_2)$ *and*
— $t' = t_\mathcal{I}$ *or*
— t' *is a successor of* $t_\mathcal{I}$ *w.r.t.* Q_1 *or* Q_2
$\mathcal{I}_i$ *is the instance over* $\mathcal{S}_{Q_i}$ *defined by* $\mathcal{I}_i = \mathcal{I}_{|\mathcal{S}_{Q_i}}$ *for* $i = 1, 2$.

We now restrict the notion of successors to *valid* successors w.r.t. some tuple set D. This restriction demands that the lineage of a tuple successor is fully contained in D. In practice, D corresponds to all compatible tuples (direct and indirect) and is used to ensure the correctness of our Why-Not answers. Intuitively, a tuple t in the output of a query, is a valid successor of a compatible tuple $t_\mathcal{I}$ if t is a successor of $t_\mathcal{I}$ and furthermore is constructed using only compatible tuples. The definition is illustrated in Figure 4.5(c).

Definition 4.1.3 (Valid successor)**.** *Let* Q *be a query,* $\mathcal{I}$ *be the input database instance for* Q *and assume the set of tuples* $D \subseteq \mathcal{I}$ *. A tuple* $t \in Q(\mathcal{I})$ *is a valid successor of some* $t_\mathcal{I} \in D \subseteq \mathcal{I}$ *w.r.t.* Q *if*
— t *is a successor of* $t_\mathcal{I}$ *w.r.t.* Q*, and*
— $lineage(t) \subseteq D$.
Next, $VS(Q, \mathcal{I}, D, t_\mathcal{I})$ *denotes the set of valid successors of* $t_\mathcal{I}$ *w.r.t.* Q.

Example 4.1.5. *In our running example, consider the query* Q_3 *rooted at the operator* m_{Q_3} *in Figure 4.1(c) and the input instance* $\mathcal{I}$ *in Figure 4.1(d). Assume the set of compatible tuples (w.r.t. to* WN = *A.name=Homer, , B.title= Iliad)* $D = \{Id_4, Id_2\} \cup \mathcal{I}_{|AB}$ *and consider the tuple* $Id_4 \in D$*. The output of the subquery* Q_2 *is*

$$Q_2(\mathcal{I}) = \{Id_4Id_7Id_2, Id_4Id_8Id_1, Id_5Id_9Id_3\}$$

(each output tuple is represented by the concatenation of the identifiers of the tuples in its lineage). The output tuple $Id_4Id_7Id_2$ *is a valid successor of* Id_4 *because it is a successor of* Id_4 *w.r.t.* Q_2 *and* $\mathcal{I}$ *and its lineage is included in D (i.e., the tuples* Id_4, Id_7, Id_2 *are all in D). On the contrary, the output tuple* $Id_4Id_8Id_1$ *is not a*

valid successor of Id_4 even though it is a successor of Id_4, because the tuple Id_1 is not in D.

In what follows the term successor means valid successor, unless mentioned otherwise.

The successor of compatible tuples are used for tracing compatible tuples on the query tree. Our goal is to find the nodes (i.e., subqueries) of the query tree where we lose the trace of the compatible tuples. In other words, we are in the quest of subqueries not producing successors of some compatible tuples. These subqueries are defined as *picky*, a term that was introduced in [CJ09] and denotes a subquery that 'picks out' of the result a compatible tuple.

As was the case for the definition of tuple successor, we first define picky operators and then picky subqueries w.r.t. a tuple set D and a tuple $t_{\mathcal{I}} \in D$. Intuitively, a query Q is picky w.r.t. a compatible tuple $t_{\mathcal{I}}$ if the rooting operator m_Q of Q does not produce successors of $t_{\mathcal{I}}$ while we still had some successors of $t_{\mathcal{I}}$ in the input of m_Q.

The definitions, given below, are illustrated in Figure 4.6.

Definition 4.1.4 (Picky operator)**.** *Let m be an operator, $\mathcal{I}$ be an input instance for m and $D \subseteq \mathcal{I}$ be a set of tuples. Then m is a picky operator w.r.t. D and $t_{\mathcal{I}} \in D$, if there is no valid successor t of $t_{\mathcal{I}}$ in $m(\mathcal{I})$.*

Definition 4.1.5 (Picky query)**.** *Let Q be a query over $\mathcal{S}$, $\mathcal{I}$ be an input instance for Q and $D \subseteq \mathcal{I}$ be a set of tuples. Let $t_{\mathcal{I}}$ be a tuple in D.*
Assuming that $Q=[Q_1]BOp[Q_2]$ and that $t_{\mathcal{I}} \in \mathcal{I}_1$ (the case of $t_{\mathcal{I}} \in \mathcal{I}_2$ is dual), Q is picky w.r.t. D and $t_{\mathcal{I}}$ if

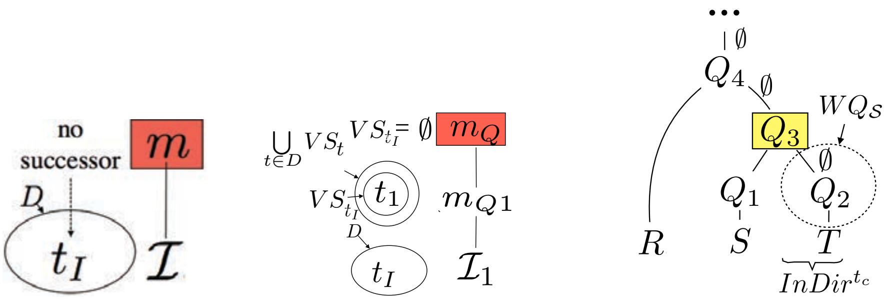
1. *$VS(Q_1, \mathcal{I}, D, t_{\mathcal{I}}) \neq \emptyset$*
2. *for each $t_1 \in VS(Q_1, \mathcal{I}, D, t_{\mathcal{I}})$, m_Q is picky w.r.t. the tuple t_1 and the set $\bigcup_{i=1,2} \bigcup_{t \in D} VS(Q_i, \mathcal{I}, D, t)$ considering the input instance $\bigcup_{i=1,2} Q_i(\mathcal{I}_i)$.*

Now, assuming that $Q=UOp[Q_1]$ and that $t_{\mathcal{I}} \in \mathcal{I}_1$, Q is picky w.r.t. D and $t_{\mathcal{I}}$ if

1. *for each $t_1 \in VS(Q_1, \mathcal{I}, D, t_{\mathcal{I}})$, m_Q is picky w.r.t. the tuple t_1 and the set $\bigcup_{t \in D} VS(Q_1, \mathcal{I}, D, t)$ considering the input instance $Q_1(\mathcal{I}_1)$*
2. *$VS(Q_1, \mathcal{I}, D, t_{\mathcal{I}}) \neq \emptyset$.*

In the definition of a picky binary query, item 1 enforces that the tuple $t_{\mathcal{I}}$ can still be traced in the input of the root operator m_Q of Q, while item 2 enforces that $t_{\mathcal{I}}$ cannot be traced in the output of m_Q. In other words, these conditions ensure us that Q is picky for $t_{\mathcal{I}}$ when we can find a valid successor of $t_{\mathcal{I}}$ in the input but not in the output of m_Q.

Here the reader should understand that we use the notion of picky query in addition to that of picky operator, to emphasize the fact that an operator is picky w.r.t. to a specific query tree. Changing the order of the operators in the query, does not guarantee that an operator remains picky or not picky.



(a) Picky operator (b) Picky query (c) Secondary Why-Not answer

Figure 4.6: Picky operator (a), picky query (b), and secondary Why-Not answer (c)

Example 4.1.6. *Consider again Figure 4.1 in page 44 and the set of compatible tuples* $D{=}\{Id_4\} \cup \mathcal{I}_{AB} \cup \mathcal{I}_B$*.* Q_1 *generates two valid successors of* Id_4*, i.e., those that are the result of joining* Id_4 *with* $Id_7 \in \mathcal{I}_{AB} \subseteq D$ *and* $Id_8 \in \mathcal{I}_{AB} \subseteq D$*, respectively. Similarly,* Q_2 *generates two valid successors of* Id_4*, their respective lineage* $\{Id_4, Id_7, Id_2\}$ *and* $\{Id_4, Id_8, Id_1\}$ *being in* D*. Finally, we observe that* Id_4 *has no (valid) successor w.r.t.* Q_3 *because* Id_4 *does not satisfy the selection condition* $A.dob > 800BC$*. Therefore,* Q_3 *is picky w.r.t.* Id_4 *and* D*.*

It is expected that for every compatible tuple, there will be one picky subquery. This is a consequence of the fact that we accept as valid only well-defined Why-Not questions, which can be translated to that there is not a valid successor of a compatible tuple in the output of the query Q. Thus, there exists for sure one picky subquery for every compatible tuple. Moreover, it holds that the picky subquery for a compatible tuple is unique, as the following property states.

Property 4.1.1. *Let* Q *be a query over* $\mathcal{S}_Q$ *and let* $\mathcal{I}$ *be an instance over* $\mathcal{S}_Q$*. Let also* $D \subseteq \mathcal{I}$ *be a set of tuples and* $t_\mathcal{I} \in D$*. Then, there exists exactly one subquery* Q' *of* Q*, s.t.* Q' *is picky w.r.t.* D *and* $t_\mathcal{I}$*.*

The proof is immediate: If there is no picky subquery for a compatible tuple $t_\mathcal{I}$, then there is a successor of $t_\mathcal{I}$ in the output of Q. This means that $Why-Notquestion$ is not valid, which does not hold. Thus there is one picky subquery for $t_\mathcal{I}$. We will show that this is unique.

Let us consider that there are two queries Q_1 and Q_2 picky for $t_\mathcal{I}$. Then, the last successor of $t_\mathcal{I}$ has to appear in the input of Q_1 and Q_2. If Q_2 is a subquery of Q_1, then by definition there is no successor of $t_\mathcal{I}$ in the output of Q_2, thus neither in the input of Q_1, which contradicts the assumption. So, Q_2 cannot be a subquery of Q_1. Otherwise (if Q_2 is not a subquery of Q_1), the last successor of $t_\mathcal{I}$ appears in two different paths (and consequently two different nodes), meaning that Q is a DAG and not a tree, which does not hold. So, there is only one picky subquery for $t_\mathcal{I}$.

Based on the notion of picky subqueries, *NedExplain* proposes three kinds of answers for a Why-Not question *WN* w.r.t. a query Q. These answers differ in terms of granularity and point of view. The main purpose behind the Why-Not answer is to indicate to the user where in the query the conditions in *WN* are violated. *WN* is composed by two parts as we have seen, WN_{conj} and WN_{α}. In the previous discussion about picky subqueries we treated the case of the conditions in WN_{conj}.

To find the subqueries of Q where the conditions in WN_{α} are violated we organise the query tree in a specific way. In order to maximize the number of subqueries for which we can verify these conditions, we organize joins such that we obtain a view V of minimal query size where $\mathcal{A}(V) \supseteq G \cup \{A_1, \ldots, A_n\}$ (see Definition 2.1.9, page 15 for a reminder of aggregate queries) and no cross product is necessary. Intuitively, V corresponds to the subquery closest to the leaf level in the query tree joining the relation where the grouped and aggregated attributes reside. We refer to V as *breakpoint* subquery. Obviously, for queries without aggregation, the condition $\mathcal{A}(V) \supseteq G \cup \{A_1, \ldots, A_n\}$ is trivially satisfied for any leaf node, i.e., for any $V \in \mathcal{I}$ (as $G \cup \{A_1, \ldots, A_n\} = \emptyset$), which results in all leaf nodes being breakpoint queries. Similarly, all leaf nodes representing relations in $\mathcal{I} \setminus \mathcal{I}_V$ can be considered as breakpoint queries. We refer to the set of all breakpoint queries, i.e., $V \cup (\mathcal{I} \setminus \mathcal{I}_V)$ as *visibility-frontier.*

Example 4.1.7. *Consider again the tree in Figure 4.1(c) and the Why-Not question* $WN = \{A.name = Homer, ap > 25\}$. *Here,* $WN_{\alpha} = \{ap > 25\}$. *From the query Q in Figure 4.1(a) we see that the group-by attribute is* `A.name` *and the aggregated-attribute is* `B.price`. *The minimum subquery containing both* `A.name` *and* `B.price` *is the subquery* Q_2. *The input instance of V is* $\mathcal{I}_V = \mathcal{I}_A \cup \mathcal{I}_{AB} \cup \mathcal{I}_B$. *So, the visibility-frontier consists only of V as there are no query input schema relations not referenced in V.*

Now, let us start by defining the *detailed* answer of a Why-Not question recording:

1. the picky query per compatible tuple (if any), and
2. in the case of aggregation, the subqueries violating the conditions on the aggregated values.

Definition 4.1.6 (Detailed Why-Not answer)**.** *Let* $(\mathcal{S}, \mathcal{I}, Q, WN)$ *be an explanation scenario. Let* Q' *be a subquery of* Q, *of the form* $UOp[Q_1]$ *if* Q' *is a unary query (respectively of the form* $[Q_1]BOp[Q_2]$ *if a binary query). Let* VF *be the visibility frontier. The detailed Why-Not answer of WN w.r.t. Q and* $\mathcal{I}$, *denoted* dW_Q, *is:*

$$\begin{aligned} \bigcup_{t_{\mathcal{I}} \in Dir} \{(t_{\mathcal{I}}, Q') \mid\ & Q' \textit{ subquery of } Q \textit{ and} \\ & Q' \textit{ picky w.r.t. } Dir \cup InDir \textit{ and } t_{\mathcal{I}}\} \\ \cup\ \{(\perp, Q') \mid\ & V \in VF \textit{ and } V \textit{ proper subquery of } Q' \textit{ and} \\ & Q_1(\mathcal{I}) (\textit{respectively } Q_1(\mathcal{I}) \cup Q_2(\mathcal{I})) \models WN_{\alpha} \textit{ and} \\ & Q'(\mathcal{I}) \not\models WN_{\alpha}\} \end{aligned}$$

The second part of this definition ensures that the conditions on aggregated attributes are verified on the input of the subquery Q', but not on its output.

Example 4.1.8. *In our running example V=Q_2. The detailed Why-Not answer for the Why-Not question WN=$\{A.name$=$Homer, ap$>$25\}$ is $\{(Id_4, Q_3)\} \cup \{Q_3\}$. Indeed Q_3 is picky w.r.t. Id_4 and $\{Id_4\} \cup \mathcal{I}_{|AB} \cup \mathcal{I}_{|B}$. Moreover, the data in the input of Q_3 satisfy WN_α (because they yield an average price of 30 $>$ 25), whereas the empty output of Q_3 does not satisfy WN_α.*

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

Definition 4.1.7 (Condensed Why-Not answer)**.** *The condensed Why-Not answer for WN w.r.t. Q and $\mathcal{I}$ is defined as $cW_Q = \{Q' | (t_\mathcal{I}, Q') \in dW_Q\}$.*

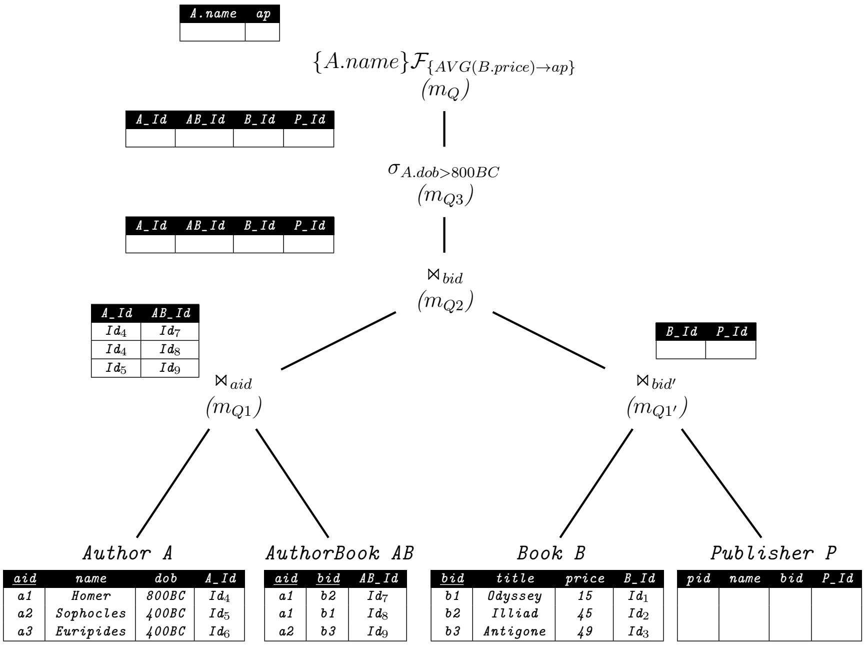
Finally, we also define a secondary Why-Not answer that takes into account the indirect compatible set $InDir$. Recall that $InDir$ includes data necessary to produce the missing answer, but that are not constrained by WN_{conj}. Consequently, the pruning of indirect compatible tuples may also be a cause for the missing-tuples, captured in the *secondary* Why-Not answer.

As a reminder, S^W is the set of relation schemas typing the tuples in Dir and $\mathcal{S}_Q - \mathcal{S}^W$ is the set of relation schemas typing the tuples in $InDir$.

Definition 4.1.8 (Secondary Why-Not answer)**.** *Let $\mathcal{S}_{sec} \in \mathcal{S}_Q - \mathcal{S}^W$. We denote by Q_{sec} the subquery of Q s.t. Q_{sec} is picky w.r.t. $\mathcal{I}$ and some $t \in \mathcal{I}_{|\mathcal{S}_{sec}}$, and for any $t' \in \mathcal{I}_{|\mathcal{S}_{sec}}$, there is no successor of t' w.r.t. Q_{sec}. Then, the secondary Why-Not answer for WN w.r.t. Q and $\mathcal{I}$ is $sW_Q = \{Q_{sec} \mid \mathcal{S}_{sec} \in \mathcal{S}_Q - \mathcal{S}^W\}$.*

Figure 4.6(c) illustrates the secondary Why-Not answer.

Example 4.1.9. *Let us add one more relation P for the publisher of the books in the example of Figure 4.1, with $\mathcal{I}_P$=$\emptyset$. Then, let us replace the right child of Q_2, i.e., B, with the subquery Q'_1=$B \bowtie_{bid'} P$. We obtain the following tree.*



A.name	ap

A_Id	AB_Id	B_Id	P_Id

A_Id	AB_Id	B_Id	P_Id

A_Id	AB_Id
Id_4	Id_7
Id_4	Id_8
Id_5	Id_9

B_Id	P_Id

Author A

aid	name	dob	A_Id
a1	Homer	800BC	Id_4
a2	Sophocles	400BC	Id_5
a3	Euripides	400BC	Id_6

AuthorBook AB

aid	bid	AB_Id
a1	b2	Id_7
a1	b1	Id_8
a2	b3	Id_9

Book B

bid	title	price	B_Id
b1	Odyssey	15	Id_1
b2	Illiad	45	Id_2
b3	Antigone	49	Id_3

Publisher P

pid	name	bid	P_Id

Now, we find that $\{Q_2\}$ is the condensed Why-Not answer w.r.t. WN and D=$\{Id_4\} \cup \mathcal{I}_{AB} \cup \mathcal{I}_B \cup \mathcal{I}_P$. However, the fact that Q_2 prunes out the compatible tuple Id_4 is not necessarily linked to the absence of join partners for Id_4 in the AB relation. On the other side, it is clear that the empty result of Q'_1 causes the result of Q_2 to be empty as well. This fact is captured by returning $\{Q'_1\}$ as secondary Why-Not answer.

4.1.5 Algorithm

Based on the framework our definitions provide, we now present *NedExplain*, an algorithm that considers an explanation scenario $(\mathcal{S}, \mathcal{I}, Q, WN)$ and outputs the detailed, condensed and secondary Why-Not answer. We limit Q to a union of SPJA queries, deferring the extension with more operators to future work. Note that the Why-Not question *WN* in the input of *NedExplain* algorithm is considered to be unrenamed. Thus, if needed, we unrename *WN* as in Definion 4.1.1.

Next, we describe *NedExplain* in two steps. First, we provide the necessary preprocessing actions computing the compatible tuples and creating the query tree. Then, we discuss the internals of the algorithm. For the rest of the discussion, recall that when we are talking about successors, we imply valid successors, unless differently stated.

Preprocessing

NedExplain starts with a preprocessing phase consisting in the steps described below.

a) CompatibleFinder. We compute the direct compatible set of tuples $Dir \subseteq \mathcal{I}$ w.r.t. *WN*, by performing appropriate `SELECT` statements that retrieve the special attribute R_Id for each involved relation R in WN_{conj} (as illustrated in Example 4.1.10). Indeed, to retrieve the compatible tuples from a relation R we execute the statement

$$\text{SELECT } R\_Id \text{ FROM } R \text{ WHERE } WN_{conj|R}$$

over $\mathcal{I}_R$, where $WN_{conj|R}$ is the conjunction of the conditions in WN_{conj}, restricted over the relation R.

In our example, we identify direct compatible tuples only in the *Author* relation, by executing the query

$$\text{SELECT } A\_Id \text{ FROM } Author\ A \text{ WHERE } A.name = \text{`}Homer'$$

Note that we require that all conditions in WN_{conj} that are specified over the same relation must be satisfied by the compatible tuples from this relation. The set of indirect tuples $InDir$ consists of all the tuples in the relations of the input query schema not appearing in WN_{conj}.

b) CanonicalQuery. A relational query Q may result in various equivalent query plans (trees) and similarly to [CJ09, CWW00], we choose a canonical query tree representation that limits the equivalent query trees to consider. The following two rationales guide our choice of canonical query tree representation that differs from the canonical tree representation of [CJ09].

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

Second, as described by Definition 4.1.6, we need to determine if a subquery (tree node) is picky and whether the associated condition(s) from WN_α is satisfied by the subquery's input and not its output. This leads to the creation of the visibility-frontier as previously discussed. Given the query tree with the visibility-frontier, we place the selections above and closest to the visibility-frontier to satisfy our first rationale. In the sequel, we denote our canonical query tree satisfying the above rationales as T.

c) Primary global structure Tab$_\mathbf{Q}$. *NedExplain* relies on one main global structure, denoted Tab_Q and used to store intermediate results. More specifically, Tab_Q contains the following *labeled* entries for each subquery of Q. Next, we refer to the subqueries of Q by their rooting operator m in the query tree T.

- *Input*: the input tuple set for m
- *Output*: the tuple set output by m
- *Compatibles*: the set of direct compatible tuples that are either in the input of m or have a valid successor in the input of m
- *Level*: the depth of m in T (the root having level 0)
- *Parent*: the parent node (a.k.a. the parent subquery) of m in T
- *Op*: the root operator of m

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

The initialization of Tab_Q is trivially done for the following entries:

- *m.Op*, based on T
- *m.Parent*, based on T
- *m.Level*, based on T
- *m.Input*=$\mathcal{I}_{|R}$, for any m associated with a base relation R
- *m.Output*=$\mathcal{I}_{|R}$, for any m associated with a base relation R
- *m.Compatibles*=$Dir_{|R}$,for any m associated with a base relation R

The rest of the entries get updated during the execution of the algorithm. In order to efficiently access the information in Tab_Q, subqueries are stored in order of decreasing depth (*m.Level*) in the query tree. We access subquery m at the position i of Tab_Q using the notation $m = Tab_Q[i]$.

d) Initialize secondary global structures. Apart from Tab_Q, we make use of

m	Input	Compatibles	Output	Level	Parent	Op
A	$\mathcal{I}_{\vert A}$	Id_4	$\mathcal{I}_{\vert A}$	4	m_{Q_1}	relation schema
AB	$\mathcal{I}_{\vert AB}$	$\emptyset$	$\mathcal{I}_{\vert AB}$	4	m_{Q_1}	relation schema
m_{Q_1}	-	-	-	3	m_{Q_2}	$\bowtie$
B	$\mathcal{I}_B$	$\emptyset$	$\mathcal{I}_B$	3	m_{Q_2}	relation schema
m_{Q_2}	-	-	-	2	m_{Q_3}	$\bowtie$
m_{Q_3}	-	-	-	2	m_{Q_4}	σ
m_Q	-	-	-	1		α

Table 4.1: Primary global structure Tab_Q upon initialization

the next global structures:

- *EmptyOutputSubQ*: contains the subqueries producing an empty result, used to determine the secondary Why-Not answer.
- *Non-PickySubQ*: contains the subqueries producing successors of compatible tuples.
- *PickySubQ*: contains the pairs (Q', blocked), where Q' is a subquery associated with the operator m and blocked=$\{t|t\in m.Input\ s.t.$ m is picky w.r.t. t and $Dir\cup InDir\}$. This structure allows us to determine the detailed and the condensed Why-Not answer.

All these structures are initially empty.

Example 4.1.10. *Consider the running example in Figure 4.1 in page 44. As discussed in Example 4.1.7, the breakpoint view V is the subquery Q_2 as this is the minimum subquery containing both `A.name` and `B.price`. The selection operator $\sigma_{A.dob>800BC}$ is then placed just above V, so as to satisfy our first rationale. Table 4.1 shows the initialization of Tab_Q given the canonical query tree of Figure 4.1(c).*

Computing the Why-Not Answer

Now we describe how we compute the Why-Not answer. Briefly, we visit the subqueries of Q bottom up on the tree T. At each subquery, we identify the successors of the compatible tuples, if any, and keep track of the picky and non-picky subqueries along the way. In addition, we identify the subqueries producing empty results and those that cause the Why-Not question condition on the aggregated attributes to fail. In the end, we return the three types of the Why-Not answer.

Algorithm 1 provides the pseudo-code for *NedExplain*. Lines 1–5 correspond to the preprocessing steps discussed above. Moreover, in line 2 we check whether *WN* is well founded or not. Remember here that for the *WN* to be well founded, the conditions over aggregated attributes are not used. The algorithm terminates without producing a Why-Not answer if the Why-Not question is not well-founded, because the data are not sufficient to provide an answer. Otherwise, *NedExplain* iterates through all the subqueries stored in Tab_Q until *checkEarlyTermination* (Algorithm 2) called at line 8 returns true or the last entry in Tab_Q has been reached.

Algorithm 1: *NedExplain*

```
   Input: Explanation scenario $(\mathcal{S}, \mathcal{I}, Q, WN)$
   Output: $Answer$, the Why-Not answer
 1 $(Dir, InDir) \leftarrow$ CompatibleFinder($WN, \mathcal{I}$);
 2 if NotWellFounded($WN_{conj}$, $(Dir, InDir)$) then
 3     return null;
 4 $T \leftarrow$ Canonical(Q);% the canonical tree of $Q$ is created and the visibility frontier is
   set %
 5 Initialize($Tab_Q$, Non-PickyOp, EmptyOutputOp, PickyOp, Dir, InDir);
 6 for (int i=1,... , #nodes in T ) do
 7     m← $Tab_Q$[i];
 8     if CheckEarlyTermination(m) then
 9         Answer.detailed←DetailedAnswer().detailed;
10         Answer.condensed←DetailedAnswer().condensed;
11         Answer.secondary←secondaryAnswer();
12         return Answer;
13     m.Output← m(m.Input);
14     p←m.Parent;
15     p.Input←p.Input ∪ m.Output ;
16     if m.Ouput=∅ then
17         EmptyOutputSubQ← EmptyOutputSubQ ∪ {$m$};
18         if m.Compatibles≠ ∅ then
19             PickySubQ ← PickySubQ ∪ {$(m, m.Compatibles)$};
20     if $m.Op \in \{\bowtie, \sigma\}$ then
21         p.Compatibles ← p.Compatibles ∪ FindSuccessors(m);
22     else
23         if m.Compatibles≠ ∅ then
24             p.Compatibles ← p.Compatibles ∪ m.Compatibles;
25             NonPickySubQ←NonPickySubQ ∪ {m};
```

Then, we compute and return the detailed Why-Not answer. Otherwise, we continue with the evaluation of the current subquery m (line 13) and update the entries of the parent p of m in Tab_Q. We also maintain the secondary global structures EmpyOutputSubQ and PickySubQ (lines 16–19). For all subqueries except for those that correspond to relation schemas, *FindSuccessors* (Algorithm 3) finds possible successors of compatible tuples in the output of the current subquery and maintains the secondary global structures PickySubQ and Non-PickySubQ (line 21). Otherwise, p.Compatibles and the global structures are updated as described in lines 22–25.

We now further discuss the sub-algorithms called by Algorithm 1.

Check for early termination (Algorithm 2). This step decides whether we have all information in hand to compute our Why-Not answer, even before reaching the root of the query tree. This can happen when at the first operator of a new level,

Algorithm 2: *CheckEarlyTermination*

```
   Input: m, a subquery
   Output: a boolean value
 1 i←position of m in $Tab_Q$;
 2 if $i \neq 1$ and m.Level$\neq$ $Tab_Q$[i-1].Level then
 3     int j=i-1;
 4     while $j \geq 1$ and $Tab_Q$[j].Level = $Tab_Q$[i-1].Level do
 5         if $Tab_Q$[j] $\in$ NonPickySubQ then
 6             return FALSE;
 7         j←j-1;
 8     while $i <$ #nodes in T do
 9         if $Tab_Q[i]$.Op=‘relation schema’ then
10             return FALSE;
11         $i \leftarrow i + 1$;
12 else
13     return FALSE;
14 return TRUE;
```

we know that there are no more compatible tuples to trace, further up the tree. To identify this case, *checkEarlyTermination* checks if m is the leftmost operator at some level in the query tree T, and then

1. we check if in the former level we have any NonPicky subqueries (Algorithm 2 lines 4–7) and if not
2. we also check if among the remaining subqueries (in this level or higher up) there exists a subquery with type ‘relation schema’ (lines 8–11), and some compatible tuples in the input of the subquery.

If according to *checkEarlyTermination*, there are no more compatible tuples to trace from the subquery m up to the root of the tree, then it returns true and Algorithm 1 terminates.

For example, consider the tree provided in Example 4.1.9, page 58. The direct compatible tuple is Id_4 from *Author* relation instance. All successors of Id_4 are eliminated at m_{Q2}. Thus, before examining m_{Q3}, the leftmost operator of level 2, the *checkEarlyTermination* algorithm returns true and *NedExplain* can continue with outputting the Why-Not answer.

Finding and managing successors (Algorithm 3). If for a subquery rooted at m *CheckEarlyTermination* returns false, we continue to computing the successors of compatible tuples in the output of m. From previous iterations, we have available the entries m.Compatibles, and m.Output. Algorithm 3 computes the valid successors in m.Output of the tuples in m.Compatibles by checking for each $o \in$ m.Output whether its lineage is in $Dir \cup InDir$ (Algorithm 3 line 3).

Any compatible tuple having at least one valid successor in the output of m, is returned by Algorithm 3 to Algorithm 1 to be added in the parent's compatibles

Algorithm 3: *FindSuccessors*

```
   Input:  m, a subquery
   Output: successors, the subset of tuples from $m.Compatibles$ having at least one
           successor in the output of $m$
1  successors$\leftarrow \emptyset$;
2  foreach $o \in m.Output$ do
3      if $lineage(o) \subseteq Dir \cup InDir$ then
4          successors $\leftarrow$ successors $\cup(lineage(o) \cap Dir)$;
5  $Blocked \leftarrow m.Compatibles \setminus successors$;
6  if $successors \neq \emptyset$ then
7      NonPickySubQ $\leftarrow$ NonPickySubQ $\cup\{m\}$;
8  if $(Blocked \neq \emptyset$ $)$ or
9  $(V$ is subquery of $m$ and $\alpha_{G,F}(m.Input) \models WN_\alpha$ and $\alpha_{G,F}(m.Output) \not\models WN_\alpha$ $)$
   then
10     PickySubQ $\leftarrow$PickySubQ $\cup$ $\{$(m,Blocked)$\}$;
11 return $successors$;
```

entry. In this case, we also add m to the set of NonPickySubQ subqueries, because after m we still can trace some compatible tuples (Algorithm 3 line 7).

Then, lines 8 through 10 in Algorithm 3 maintain the global structure PickySubQ. There are two cases, when PickySubQ is updated with a pair containing m. These cases are the conditions described in Definition 4.1.6 of detailed Why-Not answer. First, if the subquery indicated by m, is picky for a set of compatible tuples,termed *Blocked*, then PickySubQ is updated with (m,*Blocked*). Second, if the aggregation function is satisfied in the input but not in the output of m, then PickySubQ is updated with (m,$\emptyset$). Of course, the second case is applied only when possible from the schema of the subquery.

Algorithm 1 terminates by computing the three types of Why-Not answer. Algorithm 4 computes the detailed and the condensed type of answer. Algorithm 5 computes the secondary answer, which consists in the subqueries producing an empty result but are not picky subqueries.

Algorithm 4: DetailedAnswer

```
Output: $Detailed$, the detailed Why-Not answer
Detailed $\leftarrow \emptyset$;
detailed$\leftarrow \emptyset$; condensed$\leftarrow \emptyset$;
foreach $(m, Blocked) \in PickySubQ$ do
    if $Blocked = \emptyset$ then
        detailed $\leftarrow$ detailed $\cup\{(null, m)\}$;
    else
        detailed $\leftarrow$detailed $\bigcup_{t\in\text{Blocked}}\{(t, m)\}$;
    condensed $\leftarrow$ condensed $\cup\{m\}$;
return $Detailed \leftarrow \{detailed, condensed\}$;
```

Algorithm 5: Secondary Answer

```
Output: the set of subqueries having an empty result and comprising the
        secondary Why-Not answer
$secondary \leftarrow \emptyset$ ;
forall the $m$ $subqueries \in EmptyOutputSubQ$ do
    if $m \notin condensedAnswer$ then
        secondary $\leftarrow$ secondary $\cup$ {m};
return $secondary$;
```

m	m.Input	m.Output	m.Compatibles	m.Blocked
A	$\mathcal{I}_{\|A}$	$\mathcal{I}_{\|A}$	Id_4	$\emptyset$
AB	$\mathcal{I}_{\|AB}$	$\mathcal{I}_{\|AB}$	$\emptyset$	$\emptyset$
m_{Q_1}	$\mathcal{I}_{\|A}, \mathcal{I}_{\|AB}$	$Id_4 \bowtie Id_7, Id_4 \bowtie Id_8,$ $Id_5 \bowtie Id_9$	Id_4	$\emptyset$
B	$\mathcal{I}_{\|B}$	$\mathcal{I}_B$	$\emptyset$	$\emptyset$
m_{Q_2}	$Id_4 \bowtie Id_7, Id_4 \bowtie Id_8,$ $Id_5 \bowtie Id_9, \mathcal{I}_B$	$Id_4 \bowtie Id_7 \bowtie Id_2,$ $Id_4 \bowtie Id_8 \bowtie Id_1,$ $Id_5 \bowtie Id_9 \bowtie Id_3$	Id_4	$\emptyset$
m_{Q_3}	$Id_4 \bowtie Id_7 \bowtie Id_2,$ $Id_4 \bowtie Id_8 \bowtie Id_1,$ $Id_5 \bowtie Id_9 \bowtie Id_3$	$\emptyset$	Id_4	Id_4
m_Q	$\emptyset$		$\emptyset$	

Table 4.2: Tab_Q after executing *NedExplain* with the running example.

Example 4.1.11. *Continuing the Example 4.1.10, Table 4.2 represents a complete version of Tab_Q after the termination of the* NedExplain *Algorithm. However for convenience, the fields for Level, Parent and Op that have been already determined in the pre-processing phase are omitted in Table 4.2. Briefly, in each iteration in the main for-loop of Algorithm 1, i.e., for each subquery in Tab_Q, the respective sub-*

query entry has been updated. For a clarification on the generated results, consider the following cases:

- *row A: This row corresponds to the Author relation.*

  Since A is the first node in Tab_Q, Algorithm 2 does not allow for an early termination. The output entry for A is set to $\mathcal{I}_A$ in Algorithm 1. The parent subquery is m_{Q_1}, so the entries m_{Q_1}.Input and m_{Q_1}.Compatibles entries are filled in with A.Output and A.Compatibles, respectively. Then, A is added in the NonPickySubQ structure, because it contains (direct) compatible tuples.

- *row m_{Q_3}: This row corresponds to the subquery $m_{Q_3} = \sigma_{A.dob>800BC}[m_{Q_2}]$.*

  Algorithm 1 computed the values for the entries of the previous rows of the table in previous iterations. In addition, the current row's m_{Q_3}.Input and m_{Q_3}.Compatibles were also filled in with the values of m_{Q_2}.Output and m_{Q_2}.Compatibles $\setminus$ m_{Q_2}.Blocked respectively. In more detail, m_{Q_3}.Compatibles consists of the tuple Id_4 as there are two valid successors of this compatible tuple in the output of m_{Q_3} (the tuples $Id_4 \bowtie Id_7 \bowtie Id_2$ and $Id_4 \bowtie Id_8 \bowtie Id_1$). Algorithm 2 does not allow for an early termination, since m_{Q_2} (the only former level subquery) is not picky. Algorithm 1 proceeds with the evaluation of m_{Q_3} on its inputes and fills the entries m_{Q_3}.Output, and the parent's m_Q.Input accordingly. Then, we continue with the computation of the successors. In Algorithm 3 line 8 the first condition is satisfied: m_{Q_3} has blocked all the compatible tuples in m_{Q_3}.Compatibles (i.e., m_{Q_3}.Blocked=m_{Q_3}.Compatibles). The second condition is undecidable, because the output result set of m_{Q_3} is empty. However, since m_{Q_3} is a picky subquery, the pair $(m_{Q_3}, \{Id_4\})$ is added in the PickySubQ structure. At this stage, the state of NonPickySubQ is $\{A, AB, m_{Q_1}, B, m_{Q_2}\}$ and of PickySubQ is$\{(m_{Q_3}, \{Id_4\})\}$.

- *row m=m_Q: This row corresponds to the root operator of the query tree, thus to query Q. m_Q corresponds to the first tree node having Level=1. Since m_{Q_3}, which is the only subquery in the previous level, is a picky subquery and there no subqueries higher on the tree that could contain some compatible tuples, Algorithm 2 returns true. In this case, the Algorithm 1 enters the termination mode and computes the Why-Not answer. In more detail, Algorithm 4 returns the detailed Why-Not answer: $\{(Id_4, m_{Q_3})\}$ and the condensed Why-Not answer: $\{m_{Q_3}\}$. Algorithm 5 returns an empty secondary Why-Not answer.*

Extention to union queries and general Why-Not questions. Previously, we discussed the *NedExplain* Algorithm assuming one explanation scenario $(\mathcal{S}, \mathcal{I}, Q, WN)$, where the query Q is an SPJA query and the Why-Not question is one set of conditions over the output schema Γ_Q, captured in *WN*.

To extend the algortihm for the case of a union of SPJA queries:

1. We split the query Q to two sub-queries Q_1 and Q_2, one for each part of the union.
2. The unrenaming of the Why-Not question *WN* results to two Why-Not questions WN_1 and WN_2, each one corresponding to one of the sub-trees Q_1 and Q_2.

3. We run Algorithm 1 for each explanation scenario $(\mathcal{S}_i, \mathcal{I}_i, Q_i, WN_i)$ for $i = 1, 2$.
4. The final Why-Not answer is the union of the Why-Not answers obtained from the separated cases.

Similarly, we proceed for more than one unions.

Alternatively, we can treat the union as any other operator on the query tree. However, we know a priori that union operators are not picky, as they do not eliminated any tuples.

As far as negation is concerned, the situation is a little more complicated, as the solution would require to combine why-not provenance (for the left subquery of the negation (difference) operator) with why provenance (for the right subquery). If we want to restrict our algorithm to producing query-based explanations, then the extension is rather simple: we proceed as usual for the left part of the query, while we consider the right part as a 'black-box'. To characterize a difference operator node as picky for some compatible tuple, it suffices to follow Definition 4.1.5, for binary operators. In this case, compatible tuples are meaningful to identify only on the source relations of the left subquery.

To extend the algorithm for the case of a general Why-Not question (Definition 2.2.2) gWN=$\{WN_1, \ldots, WN_n\}$, we run Algorithm 1 for each explanation scenario $(\mathcal{S}, \mathcal{I}, Q, WN_i)$ for i=$1, \ldots, n$. The final Why-Not answer is the union of the Why-Not answers obtained from the separated cases.

Complexity. The time complexity of the initialization steps is constant. The worst time complexity for Algorithm 2 is $O(|Q|)$, where $|Q|$ represents the number of query operators (a.k.a. query tree nodes). The worst time complexity for Algorithm 3 is $O(|\mathcal{I}_R|)$, where $|\mathcal{I}_R|$ is the maximum size of a source relation. The worst time complexity for Algorithm 4 and Algorithm 5 is $O(|Q|)$.

So the total worst time complexity of *NedExplain* is in $O(|Q|(|Q| + \mathcal{I}_R) + |Q|) = O(|Q|^2 + |Q| * |\mathcal{I}_R|)$. If we consider that the size of a relation instance is typically much larger than the size of a query then the previous time complexity becomes $O(|Q| * |\mathcal{I}_R|)$.

4.1.6 Experiments

In this section we display a comparative evaluation of our algorithm with respect to the *Why-Not* algorithm [CJ09]. Briefly, the *Why-Not* algorithm [CJ09] identifies a set of *frontier picky manipulations* that are responsible for the exclusion of missing-answers from the result by tracing *unpicked data items* (tuples) through the workflow. Two alternatives are proposed for traversing the workflow: a bottom-up approach and a top-down approach. The main difference between the two approaches lies in the efficiency of the algorithms (depending on the query and the Why-Not question). In [CJ09], it is stated that both approaches are equivalent as they produce the same set of answers. We have implemented *NedExplain* and *Why-Not* (actually, its bottom-up version as it most resembles the approach of *NedExplain*) using Java, based on source code kindly provided by the authors of *Why-Not*. The original *Why-Not* implementation, as well as ours, relies on the lineage tracing provided by Trio (http://infolab.stanford.edu/trio/). We ran the experiments on an Oracle Virtual Machine running Windows 7 and using 2GB of main memory of a Mac Book Air with 1.8 GHz Intel Core i5, running MAC OS X 10.8.3. We used PostegreSQL 9.2 as database.

Scenarios

We evaluate *NedExplain* and compare it to the *Why-Not* algorithm based on a number of explanation scenarios. By definition, each scenario contains a database (schema and instance), a query and a Why-Not question.

We use three databases:

1. The *crime* database. It corresponds to the sample crime database of Trio (http://infolab.stanford.edu/trio/) and was previously used to evaluate *Why-Not*. The data describe crimes and involved persons (suspects and witnesses).
2. The *imdb* database. It is built on real-world movie data extracted from IMDB (http://www.imdb.com) and MovieLens (http://www.movielens.org), describing movies, their places and their ratings.
3. The *gov* database. It is built on real-world data collected at http://bioguide.congress.gov, http://usaspending.gov, and http://earmarks.omb.gov. It contains information about US congressmen and financial activities.

The size of the relations in the databases ranges from 89 to 9341 records, with *crime* being the smallest and *gov* the largest database. For abbreviation, in the following discussion each relation instance is referred to by its initials, for example M refers to the Movies instance and L to the Locations instance. Moreover, when multiple instances of some relation are used, we refer to them using numbers, e.g., $M1$ and $M2$ for two instances of the relation M.

Based on the databases, we have built a series of queries and Why-Not questions to complete the scenarios. The queries have been designed to include simple (Q4, Q6) and more complicated (Q1, Q3, Q5, Q7) queries, queries containing self-joins (Q3, Q4), queries having empty intermediate results (Q2), SPJA queries (Q8, Q9) and SPJU queries (Q12). The queries are displayed in relational algebra in Table 4.3. The scenarios are described in Table 4.4. To pinpoint the differences between the two algorithms, some scenarios consider the same query with a different Why-Not question.

Next, we evaluate *NedExplain* and *Why-Not* both in terms of answer quality and efficiency using these predefined scenarios.

Answer Quality

Table 4.5 summarizes the Why-Not answers obtained by processing our scenarios with *NedExplain* and *Why-Not* algorithms. For *NedExplain*, we distinguish among

Query	**Expression**
Q1	$\pi_{P.name,C.type}[C \bowtie_{sector} W \bowtie_{witnessName} S \bowtie_{hair,clothes} P]$
Q2	$\pi_{P.name,C.type}[\sigma_{C.sector>99}[C] \bowtie_{sector} W \bowtie_{witnessName} S \bowtie_{hair,clothes} P]$
Q3	$\pi_{W.name,C2.type}[W \bowtie_{sector2} C2 \bowtie_{sector1} \sigma_{C1.type=Aiding}[C1]]$
Q4	$\pi_{P2.name}[P2 \bowtie_{!name,hair} \sigma_{P1.name<B}[P1]]$
Q5	$\pi_{name,L.locationid}[L \bowtie_{movieId} \sigma_{M.year>2009}[M] \bowtie_{name} \sigma_{R.rating\geq 8}[R]]$
Q6	$\pi_{Co.firstname,Co.lastname}[\sigma_{AA.party=Republican}[AA] \bowtie_{id} \sigma_{Co.Byear>1970}[Co]]$
Q7	$\pi_{SPO.sponsorId,SPO.sponsorln,E.camount}[E \bowtie_{eId} \sigma_{ES.sub=Sen.\ Com.}[ES] \bowtie_{id} \sigma_{SPO.party=Rep.}[SPO]]$
Q8	$\alpha_{\{P.name\},\{count(C.type)\rightarrow ct\}}[\sigma_{sector>80}[C \bowtie_{sector} W \bowtie_{witnessName} S \bowtie_{hair,clothes} P]]$
Q9	$\alpha_{\{SPO.sponsorln\},\{sum(E.camount)\rightarrow am\}}[$ $\sigma_{substage=Senate\ Committee}[\sigma_{party=Republican}[E \bowtie_{earmarkid} ES \bowtie_{id} SPO]]]$
Q10	$\pi_{Co.lastname}[Co \bowtie_{id} \sigma_{AA.state=NY}[\sigma_{AA.party=Democrat}[AA]]]$
Q11	$\pi_{SPO.sponsorln}[\sigma_{SPO.state=NY}[\sigma_{SPO.party=Democrat} SPO]]$
Q12	$Q10 \cup_v Q11$

Table 4.3: Queries for experiments

the detailed, the condensed and the secondary Why-Not answer, as defined by Definitions 4.1.6–4.1.8.

At first sight, the answers provided by *Why-Not* are simpler and clearer; they generally consist of a small number of subqueries. On the other hand, *NedExplain* provides answers more complex in structure, but more informative. Notice that the condensed answers resemble the answers returned by *Why-Not* and provide an "easily-consumable" answer as well. Still, *NedExplain* condensed answers and *Why-Not* algorithm answers are not the same, as we will see later in this section. The subsequent discussion first highlights the differences among the three types of *NedExplain* answers before we compare the results produced by *NedExplain* with those produced by the *Why-Not* algorithm.

Detailed vs. condensed and secondary Why-Not answers. Consider the *Crime6* scenario associated to the query tree for $Q3$ depicted in Figure 4.7(b). The condensed answer indicates that m_8 is a picky subquery. The detailed answer consists of pairs of the form $(Id_i, m_8), i = 1, \ldots, 11$. For this case, the detailed answer does not provide substantial new insights compared to the condensed answer, as m_8 is responsible for pruning all compatible tuples. In this situation, the condensed answer is the most appropriate to return to the user.

However, in other cases, the simplicity of the condensed answer may hide from the user more specific, but essential information, e.g., in cases where the answer is not a single subquery. For instance, in *Crime7*, the condensed answer identifies m_8 and m_9 as picky subqueries (see $Q3$ in Figure 4.7(b)). From the detailed answer, we moreover obtain the knowledge that there were eleven tuples (originating from the $C2$ relation) for which m_8 is picky, but also one tuple (originating from W) for which m_9 is picky. This information can be useful, as it indicates that no valid successors of compatible crime tuples reached m_9 to join with valid witness tuples. So, the existence of a more detailed answer can be of major help towards understanding the

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)
Crime9	Q8	((P.Name:Betsy,ct:x),x>8)
Crime10	Q8	(P.Name:Roger)
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)
Gov6	Q9	((name:Bennett,am:x),x=18700)
Gov7	Q12	(name:JOHN)

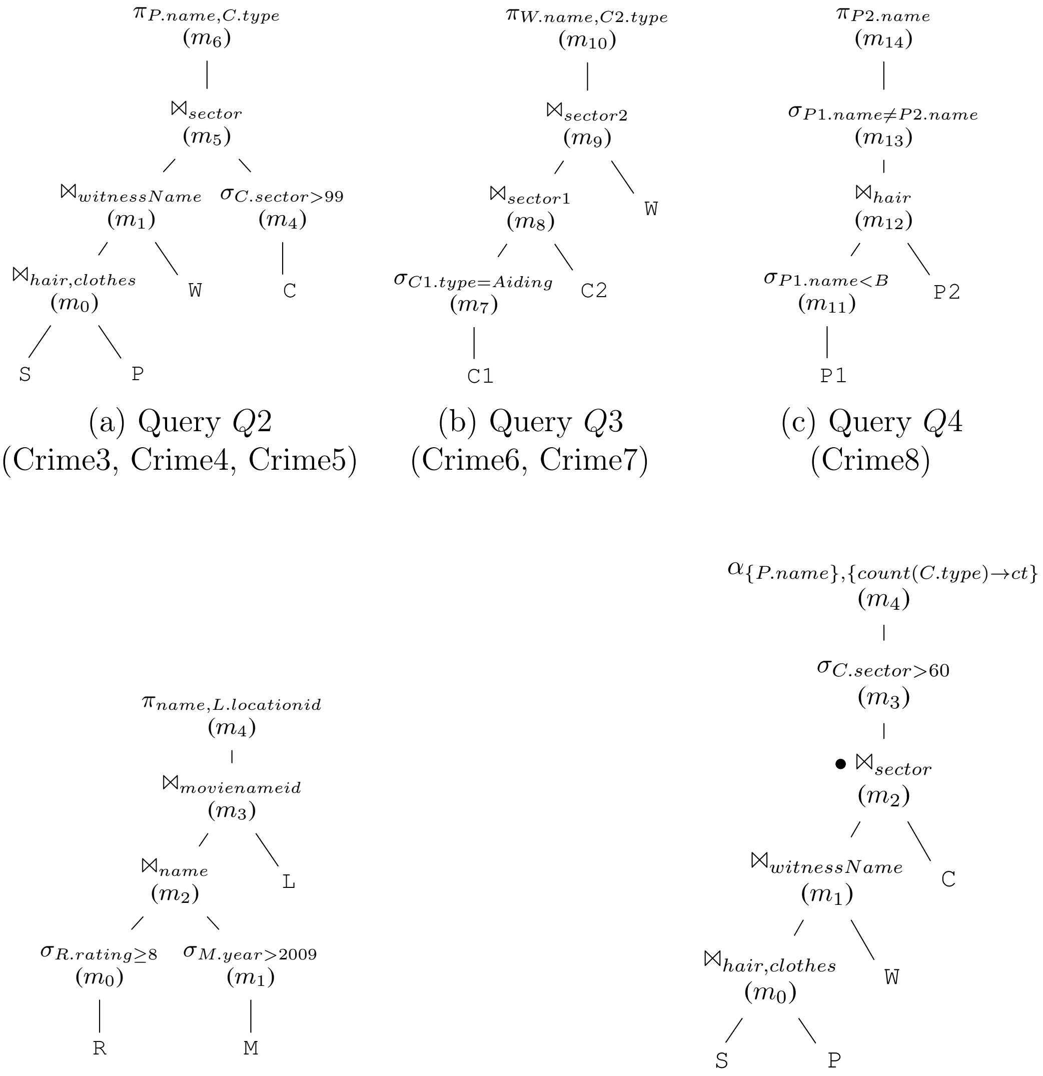
Table 4.4: Scenarios

provided explanation.

***NedExplain* vs. *Why-Not*.** Our first comparison between *NedExplain* and *Why-Not* focuses on the $Crime5$ scenario with the associated query $Q2$, whose query tree is given in Figure 4.7(a). $Q2$ has an intermediate empty result on m_4 ($\sigma_{sector>99}(C)$). The *Why-Not* algorithm identifies m_4 as a picky subquery, which is technically correct considering the definitions in *Why-Not*. Yet, m_4 is not pruning itself any direct compatible tuples, but rather destroys any indirect compatible tuples that could potentially join with direct compatible tuples. This fact is captured by classifying m_4 in the *NedExplain* secondary answer.

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

Another problem having its origin in the identification of compatible tuples can


$\pi_{P.name,C.type}$ (m_6)
$\bowtie_{sector}$ (m_5)
$\bowtie_{witnessName}$ (m_1)
$\sigma_{C.sector>99}$ (m_4)
$\bowtie_{hair,clothes}$ (m_0)
W
C
S
P
$\pi_{W.name,C2.type}$ (m_{10})
$\bowtie_{sector2}$ (m_9)
$\bowtie_{sector1}$ (m_8)
W
$\sigma_{C1.type=Aiding}$ (m_7)
C2
C1
$\pi_{P2.name}$ (m_{14})
$\sigma_{P1.name\neq P2.name}$ (m_{13})
$\bowtie_{hair}$ (m_{12})
$\sigma_{P1.name<B}$ (m_{11})
P2
P1
$\alpha_{\{P.name\},\{count(C.type)\rightarrow ct\}}$ (m_4)
$\sigma_{C.sector>60}$ (m_3)
● $\bowtie_{sector}$ (m_2)
$\bowtie_{witnessName}$ (m_1)
C
$\bowtie_{hair,clothes}$ (m_0)
W
S
P
$\pi_{name,L.locationid}$ (m_4)
$\bowtie_{movienameid}$ (m_3)
$\bowtie_{name}$ (m_2)
L
$\sigma_{R.rating\geq 8}$ (m_0)
$\sigma_{M.year>2009}$ (m_1)
R
M

(a) Query $Q2$ (Crime3, Crime4, Crime5)

(b) Query $Q3$ (Crime6, Crime7)

(c) Query $Q4$ (Crime8)

(d) Query $Q5$ (Imdb1, Imdb2)

(e) Query $Q8$ (Crime9, Crime10)

Figure 4.7: Query trees for queries $Q2$, $Q3$, $Q4$, $Q5$, $Q8$. The bullet at $Q8$ marks the breakpoint view V.

		***NedExplain* Answers**		
Scenario	*Why-Not*	Detailed	Condensed	Secondary
Crime1	$\emptyset$	$(P.Id{:}2, m_2)$,$(C.Id{:}2, m_2)$	m_2	$\emptyset$
Crime2	m_0	$(P.Id{:}604, m_0)$,$(C.Id{:}2, m_2)$	m_0, m_2	$\emptyset$
Crime3	m_0,m_4	$(P.Id{:}604, m_0)$,$(C.Id{:}2, m_4)$	m_4, m_0	$\emptyset$
Crime4	m_4	$(P.Id{:}2, m_5)$,$(C.Id{:}2, m_4)$	m_4, m_5	$\emptyset$
Crime5	m_4	$\emptyset$	$\emptyset$	m_4
Crime6	m_7	$(C2.Id{:}396, m_8), (C2.Id{:}85, m_8), \ldots, (C2.Id{:}112, m_8)$	m_8	$\emptyset$
Crime7	m_7	$(C2.Id{:}396, m_8), (C2.Id{:}85, m_8), \ldots, (C2.Id{:}112, m_8), (W.Id{:}2, m_9)$	m_8, m_9	$\emptyset$
Crime8	$\emptyset$	$(P2.Id{:}51, m_{12})$	m_{12}	$\emptyset$
Crime9	n.a.	$(null, m_3)$	m_3	$\emptyset$
Crime10	n.a.	$(P.Id{:}604, m_0)$	m_0	$\emptyset$
Imdb1	m_1	$(R.Id{:}124, m_2), (M.Id{:}18, m_1)$	m_1,m_2	$\emptyset$
Imdb2	$\emptyset$	$(L.Id{:}1, m_3), (M.Id{:}4, m_3), (R.Id{:}245, m_3)$	m_3	$\emptyset$
Gov1	m_2	$(Co.Id{:}569, m_0), (Co.Id{:}1495, m_0), (Co.Id{:}1072, m_2), (Co.Id{:}772, m_0)$	m_0, m_2	$\emptyset$
Gov2	m_1	$(Co.Id{:}1072, m_2)$	m_2	$\emptyset$
Gov3	m_0	$(Co.Id{:}569, m_0)$	m_0	$\emptyset$
Gov4	m_4	$(SPO.Id{:}9, m_4), (ES.Id{:}80, m_8), (ES.Id{:}78, m_8), (ES.Id{:}79, m_8)$	m_4, m_8	$\emptyset$
Gov5	m_6	$(E.Id{:}15, m_6), (E.Id{:}324, m_6), \ldots, (E.Id{:}533, m_6), (SPO.Id{:}199, m_6)$	m_6	$\emptyset$
Gov6	n.a.	$(null, m_7)$	m_7	$\emptyset$
Gov7	n.a.	$\{(Co.Id{:}772, m_{11})\}, \{\}$	$\{m_{11}\}$,$\{\}$	$\emptyset$

Table 4.5: *Why-Not* and *NedExplain* answers, per scenario

be spotted by the $Crime8$ scenario. Even though it is based on a very simple query - $Q4$ in Figure 4.7(c) - the *Why-Not* algorithm computes no explanation at all. $Q4$ asks for persons that have the same hair as persons, whose names start with a letter before B (while not being the same person). The *Why-Not* algorithm locates the compatible tuples both in P1 and P2. The compatible tuple (with the name Audrey) from P2 does not find any join partners for m_{12} from P1 as the only candidate tuples have names starting with C or D, namely Davemonet, Chiardola, and Debye. So, m_{12} is picky for the compatible tuple coming from P2. The compatible tuple from P1 survives the selection of m_{11} and joins with the three persons with equal hair color from P2, namely Davemonet, Chiardola, and Debye. Hence, m_{12} is not picky for three successors of the compatible tuple originating from P1, and it is easy to verify the same for the remaining subqueries. Hence, *Why-Not* believes that Audrey is actually not missing from the result. *NedExplain* on the other hand correctly locates the compatible tuple only in $P2$, namely the tuple ($P2.Id$:51). As mentioned previously, m_{11} does not produce any valid successor of the compatible tuple from $P2$, making m_{12} a picky subquery.

Next, we review the $Imdb2$ scenario associated with the query $Q5$ in Figure 4.7(d). The associated Why-Not question $WN{=}\{name = ChristmasStory, L.locationId = USANewYork\}$ is not in its unrenamed form; it contains the attribute *name* that is not in $\mathcal{S}_Q$, but instead was introduced through the join renaming $\nu{=}$ $(R.moviename, M.moviename, name)$ associated with the rooting operator of m_2. Thus, we first unrename WN to $WN'{=}\{R.moviename{:}ChristmasStory,$ $M.moviename{:}ChristmasStory, L.locationId{:}USANewYork\}$. Next, *NedExplain* proceeds with the computation of valid successors of the compatible tuples w.r.t. the subqueries of Q_5. This leads to the identification of m_3 as a picky subquery; after m_3 there are no more valid successors of the two direct compatible tuples. *Why-Not* on the contrary relies on tracing successors (not necessarily valid) of the compatible tuples, which in this case can be found in the result. So, *Why-Not* identifies no picky subqueries and as a result does not returns any explanation.

Let us now focus on $Crime9$, based on the SPJA query $Q8$ (see Figure 4.7(e)). As explained in Section 4.1.5, *NedExplain* identifies as breakpoint - marked by a bullet on the tree representation - the subquery $m2$ and the operators of the query are executed in a predefined order (putting selections as close as possible to the breakpoint subquery). In this way, we are able to identify $m3$ as a Why-Not answer; the condition of the Why-Not question of this scenario, $ct{>}8$ is satisfied by the input of $m3$ ($am{=}13$) but not by its output ($ct{=}7$). Thus, the detailed answer is $m3$ associated with a null value, since $m3$ is not picky in a strict sense (valid successors of $P.id$:604 still exist in the output of $m3$). A variation of this scenario is $Crime10$, where we obtain the detailed answer $(P.id{:}604, m_0)$; m_0 does not produce any valid successors of the compatible tuple $P.id$:604. Note that in this case *Why-Not* results are not available (marked by *n.a.* in Table 4.5) as the algorithm does not support aggregation.

Overall, with respect to answer quality, we observe that *NedExplain* produces correct answers as opposed to *Why-Not* that returns imprecise, or incomplete results in some cases. Furthermore, the different types of *NedExplain* answers convey more information than answers returned by the *Why-Not* algorithm, potentially improving the developer's analysis and debugging experience in the context of relational queries. To summarize the highlighted shortcomings, we present the following list:

- **Inaccurate selection of unpicked data.** The selection of the source (compatible) data items to be traced in the workflow does not properly take into account self-join, possibly leading to no explanation or wrong explanations. Furthermore, the definition of compatible data in [CJ09] excludes data that are in the lineage of some result tuple. Thus, there are cases where explanations are not produced because not all the compatible data are identified and traced. *Crime7* and *Crime8* are scenarios illustrating this.

- **Inaccurate definition of successors.** The compatible tuples are traced independently from each other based on the definition of successors instead of valid successors, as introduced in *NedExplain*. In this way, *Why-Not* inaccurately identifies subqueries to be responsible (or fails to identify them as responsible) for the missing-answers, as scenarios *Crime5* and *IMDB2* show.
- **Insufficient detail.** *Why-Not* returns a set of subqueries as explanation for the missing answer. This information could be difficult to be reused by a developer in a query-fixing trial,with no further details, such as the compatible tuples pruned by the subqueries, or if the query has empty intermediate results. Scenarios *Crime3* and *GOV4* demonstrate this lack of detail.

Runtime Evaluation

We first study the runtime distribution for the different phases of *NedExplain*. We then compare *NedExplain*'s runtime to the runtime of *Why-Not*.

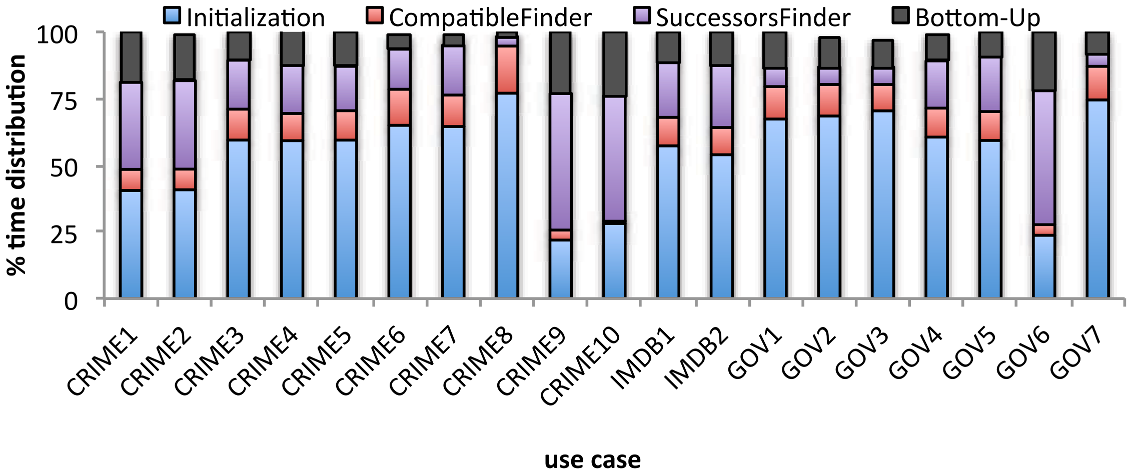


Figure 4.8: Phase-wise runtime for *NedExplain*

Phase-wise runtime. We distinguish among four phases of *NedExplain*:

1. *Initialization*: the global structures initialization
2. *CompatibleFinder*: the computation of the compatible tuples set
3. *SuccessorsFinder*: the computation of successors of compatible tuples for each subquery output, corresponding to Algorithm 3 and
4. *Bottom-up traversal*: the main *NedExplain* steps (including SQL query executions) following the bottom-up approach, i.e. Algorithm 1 without initialization and successors computation (call to Algorithm 3).

Figure 4.8 reports the distribution of the execution time over these phases for each of our scenarios. We make the following remarks.

- We observe a similar distribution for scenarios with the same query, e.g., *Crime1* and *Crime2* or *GOV1*, *GOV2*, and *GOV3*, due to the fact that these scenarios engage the same database instances and evaluate (almost) the same operators.
- For scenarios involving different queries, we observe that in general, for SPJ queries, the overall runtime is dominated by the initialization phase (between 40 and 77%), essentially caused by the initialization of all relevant java objects.
- After initialization, SuccessorFinder has the second largest impact on the overall runtime for most SPJ scenarios, with *Gov1* – *Gov3*, based on $Q6$, and *Crime8* being the exceptions. This phase essentially corresponds to Algorithm 3, which computes the lineage of tuples and performs tuple set comparisons. Let us focus for example on one of the exceptions: *Crime8*. Here the last valid successor is lost very early (on m_{12} in Figure 4.7(c)) after evaluating simple operators, which explains both the low fraction of runtime used to find successors but also the short time needed for the bottom-up traversal.
- The picture changes when considering SPJA queries, where most of the time is dedicated to the computation of valid successors. This can be explained, by the extra computations needed in the SPJA case of Algorithm 3. These computations basically require additional SQL query executions, on the input and output of tree nodes after the breakpoint view node.

Runtime comparison to *Why-Not*. Figure 4.9 displays, for each scenario, the time (in ms) each algorithm needs to produce its Why-Not answers. Generally, we observe that *NedExplain* is faster compared to *Why-Not*. One reason is that the implementation of the *Why-Not* algorithm uses Trio for lineage calculation, adding a substantial overhead to runtime especially when many trio tables are referenced as in $Crime1$ and $Crime2$. *NedExplain* traces the compatible tuples by issuing queries directly to the underlying Postgres database based on their unique identifiers in order to find their successors, which speeds up the process. However, even if we assumed that a smarter implementation of computing lineage was adopted by *Why-Not*, the run-time performance is expected to be comparable to that of *NedExplain* as the algorithms' complexity is the same.

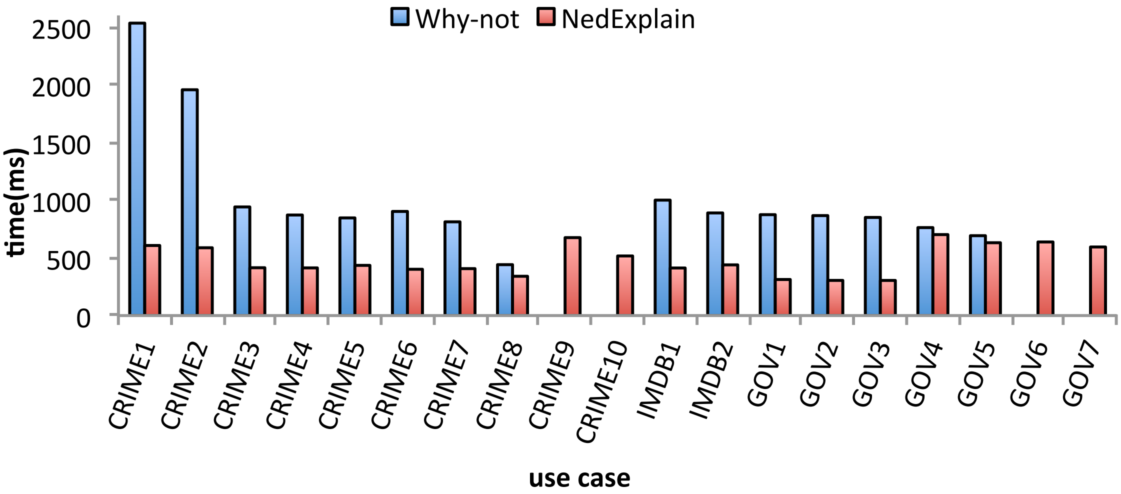


Figure 4.9: *Why-Not* and *NedExplain* execution time

4.1.7 Conclusion

We have addressed the issue of answering Why-Not questions by first formally defining, for the first time, the concepts of Why-Not question and Why-Not answer (a.k.a. query based explanations) w.r.t. relational queries including projection, selection, join, union, and aggregation (SPJUA queries). Based on these definitions, we have described *NedExplain*, an algorithm to produce correct and complete query based explanations w.r.t. the definitions, based on a specific query tree representation. As discussed and validated through experiments, *NedExplain* is capable of providing a more relevant and correct set of answers compared to the state-of-the-art *Why-Not* algorithm, while being competitive or more efficient in terms of runtime. The set of query based explanations could further be used to obtain modification-based explanations and/or in combination with instance based explanations to compute *hybrid* explanations to Why-Not questions or improve the query-based explanations part in [Her15].

In the future, an obvious optimization of *NedExplain* can be achieved by considering only the compatible tuples while tracing them on the query tree. This could be achieved by restricting the considered relation instances (on leaves of the query tree) to only compatible tuples. In this way, all tuples generated by intermediate operator nodes are guaranteed to be valid successors and less operations and storage space are required.

Besides the optimization of NedExplain, an open issue is to extend the framework to cover the whole class of relational queries, i.e., adding set difference to the relational operators. This would require tracing data that are expected to be found in the query result (compatible tuples) but also data that are not expected to be in the result (in order not to eliminate the compatible tuples).

A limitation of the query-based explanations provided by *NedExplain* is that they are dependent on the chosen query tree. For this reason, different trees can lead to different query-based explanations, which could moreover be incomplete as the process terminates at the node where the last compatible tuple is traced. Thus, the remaining nodes higher in the tree are left unexamined. This is an issue strongly related to the tree representation of queries at the basis of the *NedExplain* algorithm. In the next section, we propose a new approach towards answering Why-Not questions in a tree-independent way.

4.2 Ted

The previous section was dedicated to the *NedExplain* algorithm, the first approach developed in this thesis to answer Why-Not questions. The computed query based explanations consisted in the set of subqueries in the query Q that are marked as responsible for pruning out relevant data (a.k.a. compatible tuples). *NedExplain*, as well as the state of the art *Why-Not* algorithm (see Section 3.2) rely on a specific query tree and thus produce explanations that vary for reordered query trees. This fact leads to an open question regarding a method that can produce the same explanations regardless the ordering of the operators in the query tree. The desiderata moreover require that the produced explanations contain all possible query conditions that are to blame for not obtaining the missing answer. In addition to knowing which conditions are to blame (named picky in *NedExplain* as we saw in Section 4.1), it would be useful for the user to know in what combinations the responsible conditions should be considered. This could help in a subsequent query refining phase as the user should be aware for example that changing only the picky selection will not make a difference if she does not change also the picky join.

The following example provides an intuition on the shortcomings of the previous algorithms.

Example 4.2.1. *Consider the data of Figure 4.10 describing airlines and countries they serve and the SQL query asking about the countries served by 'old' airlines. Assume that a developer (or user) wants an explanation for the absence of Emirates from the query result as she believed that this airline company is an old established one. This question can be expressed as the Why-Not question {airline=Emirates}.*

Figure 4.11 shows two possible query plans for the SQl query. On each tree, we have marked the operators (tree nodes) computed as picky by Why-Not *[CJ09] (○) and* NedExplain *[BHT14c] (⋆) as well as the tree operators returned as part of hybrid explanations*[2] *by* Conseil *[Her13, Her15] (•). It is clear that each algorithm returns a different result for each of the two query trees. We can further observe that in most of the cases the explanation returned is only a partial result. A complete explanation should include* both *the selection that is too strict for the compatible tuple* $(Emirates, 1985, 3)$ *in the Airline table and the join because the compatible tuple does not find join partners in the Country table.*

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

Airline

airline	year	ccode
KLM	1919	1
Qatar	1993	1
Aegean	1987	2
Emirates	1985	3

Country

code	country
1	*Australia*
2	*France*

Figure 4.10: Example query and data

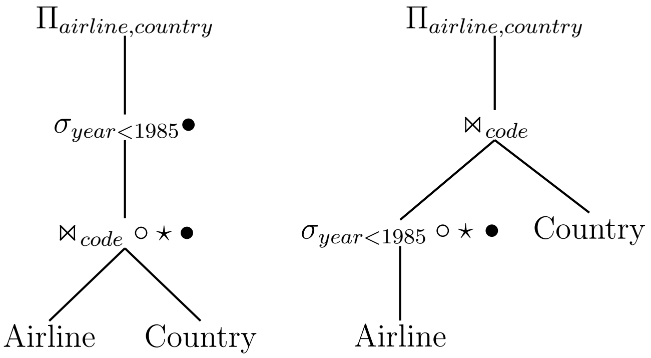


Figure 4.11: Reordered query trees for the query of Figure 4.10 and algorithm results (*Why-Not* $\circ$, *NedExplain* $\star$, *Conseil* $\bullet$)

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 some explanation is complete, and if there are other explanations that she could consider instead.

To overcome these problems we propose a new formalization of query based explanations as a *polynomial* of query conditions. Intuitively, a term of the polynomial captures one possible combination of picky conditions that should be considered together. For the previous example, $\bowtie_{code}\ \sigma_{year<1985}$ is a term capturing the fact that the join and the selection operators are together picky for the compatible tuple. Then, the different terms of the polynomial provide the user with all the possible explanations and thus enables her to choose among the different debugging possibilities. For example, consider the query in Figure 4.10, whose condition $year < 1985$ is replaced by $year < 1987$ and consider the Why-Not question $(airline \leq' Emirates')$. Now, we have two alternative possible explanations are two: (1) the selection on year is too strict for the compatible tuple with the value *Aegean*, (2) the compatible tuple with the value *Emirates* did not find any join partners. These explanations result into two different terms in the polynomial, i.e., $\bowtie_{code}\ + \sigma_{year<1985}$ indicating that they can be considered independently from one another. Except for the terms of the polynomial, the (integer) coefficient of a term provides useful insight on the estimated number of missing tuples that will be potentially returned if the term is changed. For example, the addend $1 * \sigma_{year<1985}$ means that changing the term

[2] More specifically, the query based part of the hybrid explanations is shown on the tree.

$\sigma_{year<1985}$ will yield up to one missing tuple in the result.

4.2.1 Contribution

The polynomial-based explanations approach has the following contributions:

Why-Not answer polynomial. Our formal framework is defined for conjunctive queries with inequalities and unions thereof for the relational data model under set semantics. It supports a larger class of Why-Not questions w.r.t. previous works, i.e., both simple and complex Why-Not questions. The form of the Why-Not answer is unprecedented, as this work is the first to formalize Why-Not answer polynomials providing fine-grained query based explanations. Intuitively, each term of a polynomial represents one combination of the query conditions that *together* explain the absence of some of the missing tuples and the set of all terms covers all possible such combinations. The coefficients can be used to obtain an upper bound on the number of recoverable missing tuples when properly changing the conditions of a term.

Extended formalization of the Why-Not answer polynomial. An extended formalization of Why-Not answer polynomials is provided to cover the relational data model under bag and probabilistic semantics. This confirms the robustness of the chosen polynomial representation, making it a good fit for a unified framework for representing query based explanations for different semantics.

Equivalent queries w.r.t. the Why-Not answer polynomial We define the class of equivalent queries w.r.t. the Why-Not answer polynomial. We show that isomorphic queries have isomorphic Why-Not answer polynomials given a fixed Why-Not question. This leads us to state that for a given explanation scenario the Why-Not answer polynomial is invariant of the topology of the query tree and that moreover the Why-Not answer polynomial subsumes the *NedExplain* condensed Why-Not answer. Additionally, we show that equivalent queries in general are not equivalent w.r.t. the Why-Not answer polynomial and as such, no homomorphisms exist mapping the Why-Not answer polynomials of two equivalent queries to one another. Finally, we provide an approximate Why-Not answer polynomial for a query Q, when there is available one equivalent query Q' resulting from tableau minimization of Q.

Naive *Ted* and optimized *Ted++* algorithms. We first provide a naive algorithm for computing Why-Not answer polynomials, named *Ted*. This algorithm is a straightforward implementation of the formal definitions. We show that *Ted* is impractical. We thus propose an optimized algorithm, *Ted++*, capable of efficiently computing the Why-Not answer polynomial, relying on schema and data partitioning (allowing for a distributed computation) and advantageous replacement of expensive database evaluations by mathematical calculations.

Experimental validation. We experimentally evaluate the quality of the proposed Why-Not answer polynomial and the efficiency of the *Ted++* algorithm. The experiments include a comparative evaluation to existing algorithms computing query-based explanations for SQL queries in terms of explanation quality and run-time, as well as a thorough study of the *Ted++* performance w.r.t. different parameters.

4.2.2 Preliminaries

In this section (Section 4.2) we consider conjunctive queries with inequalities and unions thereof (Definition 2.1.2) and a set of simple or complex Why-Not questions gWN (Definition 2.2.1). Initially we restrict our discussion to the relational data model under set semantics. An extension to other data model semantics (bag and probabilistic) is provided in Section 4.2.7.

In the following we focus our discussion on a single query Q and a single Why-Not question WN. Extending the method for covering unions of conjunctive queries and general Why-Not questions is trivial and is discussed in the end of this chapter.

As usual, to consider a Why-Not question and proceed to answer it, the Why-Not question should be well founded (Definition 2.2.4). Under this assumption, a compatible tuple τ is defined as in Definition 2.2.5 and the set of compatible tuples is denoted by CT.

The following example serves as the running example summarizing the preliminary notions for the problem and setting the basis of the subsequent discussion.

Example 4.2.2. *Figure 4.12 describes the explanation scenario of the running example for this section. Figure 4.12(a) displays an instance $\mathcal{I}$ over the schema $\mathcal{S}$=$\{R,S,T\}$. Figure 4.12(b) displays a query Q over $\mathcal{S}$, whose conditions have been named $op_1,\ldots,op_5$ for convenience. $R.B$=$T.B$ and $T.D$=$S.D$ are complex conditions, whereas the others are simple. Moreover, the query result is*

$$Q[\mathcal{I}]=\{(R.B{:}5, S.D{:}4, T.C{:}9)\}$$

Then, we may wonder why in $Q[\mathcal{I}]$ there is no tuple for which $R.B<S.D$ and $T.C \leq 9$. According to Definition 2.2.1, this Why-Not question can be seen as the set of the conditions $\{R.B<S.D, T.C \leq 9\}$ (Figure 4.12(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 $Q_{WN}=\sigma_{R.B<S.D \wedge T.C \leq 9}[R \times S \times T]$, and contains 12 tuples. For example, one compatible tuple is

$$\tau_1=(R\_Id{:}Id_1, R.A{:}1, R.B{:}3, S\_Id{:}Id_5, S.D{:}4, S.E{:}8, T\_Id{:}Id_8, T.B{:}3, T.C{:}4, T.D{:}5)$$

R

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

S

D	E	S_Id
4	8	Id_5
5	3	Id_6
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 Instance $\mathcal{I}$

Q

$\mathcal{S} = \{R, S, T\}$

$\Gamma = \{R.B, S.D, T.C\}$

$C = \{op_1, op_2, op_3, op_4, op_5\}$

C

op_1	$R.A > 3$
op_2	$R.B = T.B$
op_3	$T.C \geq 8$
op_4	$T.D = S.D$
op_5	$S.E \geq 3$

(b) query $Q = (\mathcal{S}, \Gamma, C)$ and naming of conditions in C

WN

{R.B<S.D, T.C <= 9}

(c) Why-Not question *WN*

Figure 4.12: Scenario of running example

In what follows we synthesize compatible tuple by providing $\mathcal{R}$_Id attributes only. In this way we write that τ_1=(R_Id:Id_1, S_Id:Id_5, T_Id:Id_8). Table 4.2.2 summarizes the set of compatible tuples for the running example.

4.2.3 Polynomial Explanations

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

Definition 4.2.1 (Explanation for τ)**.** *Let $(\mathcal{S}, \mathcal{I}, Q, WN)$ be an explanation scenario, where Q=$(\mathcal{S}, \Gamma, C)$ and WN is a well founded Why-Not question. Let CT be the set of compatible tuples w.r.t. WN and $\mathcal{I}$. Let $\tau \in CT$ be a compatible tuple. Then, the explanation for τ is the set of conditions $\mathcal{E}_\tau$=$\{op | op \in C$ and $\tau \not\models op\}$.*

Note that the conditions in $\mathcal{E}_\tau$ may be called picky conditions, in connection to the picky subqueries in *NedExplain* and picky manipulations in Why-Not [CJ09].

Table 4.6: Compatible tuples for scenario in Figure 4.2.2

CT

R_Id	S_Id	T_Id
Id_1	Id_5	Id_8
Id_1	Id_5	Id_9
Id_1	Id_5	Id_{10}
Id_1	Id_5	Id_{11}
Id_1	Id_6	Id_8
Id_1	Id_6	Id_9
Id_1	Id_6	Id_{10}
Id_1	Id_6	Id_{11}
Id_2	Id_5	Id_8
Id_2	Id_5	Id_9
Id_2	Id_5	Id_{10}
Id_2	Id_5	Id_{11}

Example 4.2.3. *Consider the compatible tuple τ_1 in Example 4.2.2. The conditions of Q (see Example 4.2.2), not satisfied by τ_1 are op_1, op_3, and op_4. So, $\mathcal{E}_{\tau_1}$=$\{op_1, op_3, op_4\}$ is the explanation for the exclusion of τ_1 out of $Q[\mathcal{I}]$. More accurately, $\mathcal{E}_{\tau_1}$ is the explanation for the exclusion of the missing tuple that could have existed in the result, if τ_1 was not pruned out. For convenience however, we simply say that $\mathcal{E}_{\tau_1}$ is the explanation for τ_1.*

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

Of course, several compatible tuples may share the same explanation. Thus, the final Why-Not answer is a polynomial having as variables the query conditions and as integer coefficients the number of compatible tuples sharing an explanation.

Definition 4.2.2 (Why-Not answer polynomial). *Let Δ=$(\mathcal{S}, \mathcal{I}, Q, WN)$ be an explanation scenario, where Q=$(\mathcal{S}, \Gamma, C)$ and WN is a well founded Why-Not question. Let CT be the set of compatible tuples w.r.t. WN and $\mathcal{I}$. The Why-Not answer polynomial w.r.t. Δ is defined as the polynomial*

$$\text{PEX} = \sum_{\mathcal{E} \in E} coef_{\mathcal{E}} \prod_{op \in \mathcal{E}} op$$

where $E = 2^C$ and $coef_{\mathcal{E}} =\mid \{\tau \in CT \mid \mathcal{E}$ is the explanation for $\tau\} \mid$.

Intuitively, E contains all potential explanations, and each of these explanations prunes from zero to at most $|CT|$ compatible tuples. Moreover, the coefficient of an explanation is equal to the number of compatible tuples with the same explanation. As a result, no coefficient could have a value greater than the number of the compatible tuples in CT as expressed in the following property.

Property 4.2.1. *Assuming the previous assumptions, and that the Why-Not answer polynomial is* $\sum\limits_{\mathcal{E}\in E} coef_{\mathcal{E}} \prod\limits_{op\in\mathcal{E}} op$ *w.r.t.* Δ*, then for each explanation* ϵ *it holds that*

$$coef_{\mathcal{E}}{\in}\{0,\ldots,|CT|\}$$

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 Definition 4.2.2 offers, through its coefficients, some useful hints to users interested in the *number* of recoverable missing tuples. More precisely, by choosing to repair the conditions in a given explanation $\mathcal{E}$, we obtain an upper bound for the number of compatible tuples that can be recovered. A change of the conditions in $\mathcal{E}$ may entail the repair of some other explanation as well, those that are strict subsets of $\mathcal{E}$. In this way, the upper bound is the sum of the coefficients of all the explanations that are sub-sets of (the set of conditions of) $\mathcal{E}$.

Definition 4.2.3 (Upper Bound)**.** *Assuming the previous assumptions, the upper bound* $\lambda_{\mathcal{E}}$ *of recoverable compatible tuples by the explanation* $\mathcal{E}$ *is*

$$\lambda_{\mathcal{E}}{=}\sum_{\mathcal{E}'\subseteq\mathcal{E}} coeff_{\mathcal{E}'}$$

Example 4.2.4. *In Example 4.2.3 we found the explanation* $\{op_1, op_3, op_4\}$*, which is translated to the polynomial term* $op_1op_3op_4$*. Taking into consideration all 12 compatible tuples of our example, we obtain the Why-Not answer polynomial:*

$$2op_1op_4 + 2op_1op_3op_4 + 4op_1op_2op_4 + 2op_1op_2op_3 + 2op_1op_2op_3op_4$$

In the polynomial, each addend, composed by a coefficient and an explanation, captures a way to obtain missing tuples. For instance, the explanation $op_1op_2op_4$ *indicates that we may recover some missing tuples if* op_1 *and* op_2 *and* op_4 *are changed. The upper bound* $\lambda_{op_1op_2op_3} = 4 + 2 = 6$ *indicates the maximum number of missing tuples that a developer may recover in the query result if she changes the conditions in the explanation* $op_1op_2op_3$*. Note, that in the above equation 4 is the coefficient of* $op_1op_2op_3$ *and 2 is the coefficient of its sub-explanation* op_1op_4*.*

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

Example 4.2.5. *Consider again the Why-Not answer polynomial* PEX$=2op_1op_4 + 2op_1op_3op_4 + 4op_1op_2op_4 + 2op_1op_2op_3 + 2op_1op_2op_3op_4$ *from the previous example. In a setting where a developer is interested in making only the least changes on the query conditions, the minimized polynomial that we could provide her with is*

$$2op_1op_4 + 2op_1op_2op_3$$

It is clear that these explanations are not included in any other explanation of PEX *and thus correspond to the minimum alternative changes that can be made to the query in order to be able to obtain some missing tuples.*

In a different situation, consider a developer interested in picking a term that if changed may lead to recovering 4 missing tuples. The coefficients inform us that she may succeed by changing the conditions in the explanation $op_1op_3op_4$ or $op_1op_2op_4$ or $op_1op_2op_3op_4$.

Finally, assume that the developer is willing to change only selection conditions in the query. The polynomial informs her that this is not possible because none of the terms is solely composed by selection conditions.

Extension to queries with unions, general Why-Not questions and *k*-Why-Not questions

Extending the framework to unions of conjunctive queries is simple. Let $\Delta = \{\mathcal{S}, \mathcal{I}, Q_u, WN_u\}$ be an explanation scenario where Q_u is a union of conjunctive queries and WN_u is expressed over the output schema of Q_u. Assume that Q_u stands also for the set of queries involved in the union. To solve this scenario, first we may need to unrename the Why-Not question so as it is expressed only over source relation attributes, in a similar way as in Section 4.1 for *NedExplain.* Thus, for each $Q \in Q_u$ we obtain one explanation scenario Δ_Q=$\mathcal{S}_Q, \mathcal{I}_Q, Q, WN_Q$ and consequently one PEX_Q w.r.t. the scenario Δ_Q. The Why-Not answer polynomial w.r.t. the scenario Δ is $PEX = \sum_{Q \in Q_u} PEX_Q$.

In the same spirit for a general Why-Not question gWN, we consider each Why-Not question WN independently and obtain one PEX_{WN} for each WN. Thus, the Why-Not answer polynomial is $PEX = \sum_{WN \in gWN} PEX_{WN}$.

Finally, the discussion on the interpretation of the polynomial coefficients and the upper bound in Definition 4.2.3 lead us to introduce an extended form of a Why-Not question being able to express questions about the number of missing tuples expected in the result. The framework can be easily extended to answer such kind of k-Why-Not question asking “Why are there not k tuples in the result with specific characteristics (described in WN)? ”.

Definition 4.2.4. *(k-Why-Not question) Given a Why-Not question WN as in Definition 2.2.1 and a positive integer k, the tuple $k - WN$=(k, WN) denotes a k-Why-Not question.*

Algorithm 6: *Ted* algorithm

```
Input: $Q$, $\mathcal{I}$, $WN$
Output: $PEX$, the Why-Not answer polynomial
Polynomial $PEX \leftarrow 0$ ;
Set $Part \leftarrow$partitioning( $\mathcal{S}$);
Set $CT \leftarrow$ CompatibleFinder($Part$, $\mathcal{I}$);
for ($\tau$: compatible tuple in $CT$) do
    $\mathcal{E}_\tau \leftarrow 1$; initialization of the explanation for $\tau$
    for (op:condition in $C$) do
        if $\tau \not\models op$ then
            $\mathcal{E}_\tau \rightarrow \mathcal{E}_\tau * op$;
    $PEX \leftarrow PEX + \mathcal{E}_\tau$;
return PEX;
```

Example 4.2.6. *Consider the Why-Not question* WN=$\{R.B{<}S.D, T.C{\leq}9\}$ *of our running example. If the developer expected to find* 6 *tuples in the result satisfying the conditions in* WN *then she could express the k-Why-Not question* $(6, WN)$*.*

If PEX is the Why-Not answer polynomial for the Why-Not question WN, then the Why-Not answer polynomial $k - PEX$ w.r.t. $k - WN$ contains only these explanations in PEX whose upper bound is lower than k. Moreover, the coefficient of an explanation $\mathcal{E}$ in $k - PEX$ is the upper bound of $\mathcal{E}$, given PEX. Thus,

$$k - PEX = \sum_{\mathcal{E} \in PEX \text{ and } \lambda_{\mathcal{E}} \geq k} \lambda_{\mathcal{E}} \mathcal{E}$$

In this way, explanations that for sure cannot lead to recovering k missing tuples, are excluded from the Why-Not answer.

Example 4.2.7. *Let us continue the previous example to find the Why-Not answer polynomial w.r.t. the k-Why-Not question* $k - WN$*. For this reason, consider given the* PEX *w.r.t.* WN *by Example 4.2.4. Then, the Why-Not answer polynomial w.r.t.* $k - WN$ *is* $6op_1op_2op_4 + 12op_1op_2Op_3op_4$*. Here, the explanations* op_1op_4 *and* $op_1op_3op_4$ *have been excluded. Indeed, the upper bounds of these explanations (2 and 4 respectively) indicate that even if we change the conditions in the explanations it is impossible to yield 6 missing tuples in the result.*

4.2.4 Ted Naive Algorithm

In this section we provide an algorithm called *Ted* that for computes Why-Not answer polynomials by a naive implementation of the previous definitions.

Algorithm 6 presents the pseudo-code for *Ted*. The first step partitions the input query schema based on the conditions of the Why-Not question. Then, *Ted* computes the set of compatible tuples CT, which is used for the computation of the Why-Not answer.

In more detail, to compute the set of compatible tuples (line 3) , we could directly perform the query Q_{WN} as indicated in Definition 2.2.5. This is however a time-consuming query as it requires cross products over the input relation instances. To improve the efficiency of the compatible tuples computation, we divide the problem into independent subproblems based on a partitioning of the relations in $\mathcal{S}$ as shown in Definition 2.2.6. As a reminder, the partitioning groups together relations that are connected through conditions of *WN*. For a simple *WN* each resulting partition contains exactly one relation, as each condition in *WN* spans over exactly one relation (this was also the case of the Why-Not questions considered in NedExplain in Section 4.1).

So, using Lemma 2.2.1, *Ted* first determines the set of partial compatible tuples for each partition and then combines the partitions using cross product to produce the set CT.

Example 4.2.8. *For the running example, the relations R, S are grouped together in one partition as they are connected through the Why-Not question condition $R.B < S.D$, whereas T forms a partition on its own. Then, the set of* partial *compatible tuples over the first partition are*

$$\{(R\_Id{:}Id_1, S\_Id{:}Id_5), (R\_Id{:}Id_1, S\_Id{:}Id_6), (R\_Id{:}Id_2, S\_Id{:}Id_6)\}$$

The set of partial compatible tuples over the second partition contains all the tuples of T. Indeed, the cross product of the partial compatible tuple sets results into the set of compatible tuples CT containing the 12 tuples shown in Table 4.2.2.

The computation of the Why-Not answer polynomial directly follows from the definitions of Section 4.2.3. Thus, lines $4-9$ describe the iteration over the set CT, the computation of the explanation for each $\tau \in CT$ (see Example 4.2.3) and the final overall Why-Not answer polynomial (see Example 4.2.4).

Complexity analysis. The three main phases of *Ted* are the partitioning phase, the computation of concatenated compatible tuples, and the computation of the Why-Not answer polynomial.

1. As the partitioning is performed on the query input schema, its worst case complexity is the arithmetic progression $a_n = 1 + (n-1) * 1 = n$, where $n = |\mathcal{S}|$. So, we have $O(|\mathcal{S}|)$ for this step.
2. The compatible tuples data complexity is bound by the cost of performing a cross product over the relation instances and thus is $O(\prod_{R\in\mathcal{S}} |\mathcal{I}_{|R}|) = O(|\mathcal{I}_{|R}|^{|\mathcal{S}|})$, where $|\mathcal{I}_{|R}|$ is the largest relation instance in the input of Q.
3. Finally, for the computation of the Why-Not answer polynomial the most complex operation needs to check all the query conditions C for all the compatible tuples, and thus the complexity is $|\mathcal{I}_{|R}|^{|\mathcal{S}|} * |C|)$.

So, the respective worst case complexities add up to $O(|\mathcal{I}_{|R}|^{|\mathcal{S}|} + |\mathcal{I}_{|R}|^{|\mathcal{S}|} * |C|)$, which simplifies to $|\mathcal{I}_{|R}|^{|\mathcal{S}|}$.

Algorithm 7: *Ted++*

```
   Input: $Q$=$(\mathcal{S}, \Gamma, C)$, $\mathcal{I}$, $WN$
   Output: $PEX$
1  $E \leftarrow$ powerset($C$);
2  $\mathcal{P} \leftarrow$validPartitioning($\mathcal{S}, WN$); * Definition 2.2.6 *
3  for $Part$ in $\mathcal{P}$ do
4      $CT_{|Part} \leftarrow (Part, \mathcal{A}(Part), WN_{|Part})[\mathcal{I}_{|Part}]$ ;
5  coefficientEstimation($E, \mathcal{P}$);
6  $PEX$ $\leftarrow$post-processing(); * Equation (F) *
7  return PEX;
```

As a conclusion, the naive implementation of the provided definitions for Why-Not answer polynomial is theoretically shown to be inefficient. As the complexity discussion shows, not only is the computation of the set of compatible tuples time and space consuming as it often requires cross product executions, but also the iteration over this (potentially very large) set is time consuming. As the experiments in Section 4.2.6 confirm, this complexity renders *Ted* inapplicable for many cases.

4.2.5 Ted Optimized Algorithm

To overcome *Ted*'s poor performance, we provide here an optimized algorithm, called *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 potential explanations, which is expected to be much smaller, (ii) computing sets of *passing partial* compatible tuples, and (iii) computing the *number* of *eliminated* compatible tuples for each explanation. Intuitively, passing tuples w.r.t. an explanation are tuples satisfying the conditions of the explanation. Finally, we compute the polynomial based on mathematical calculations.

Theorem 4.2.1 states that *Ted++* is sound and complete w.r.t. Definition 4.2.2.

Theorem 4.2.1. *Given a query q, a Why-Not question WN and an input instance $\mathcal{I}$,* Ted++ *computes exactly* PEX.

Algorithm 7 provides an outline of *Ted++*. The input consists in an explanation scenario $\mathcal{S}, \mathcal{I}, Q, WN$, where Q=$(\mathcal{S}, \Gamma, C)$. Firstly in Algorithm 7, line 1, all potential explanations (combinations of the conditions in C) are enumerated, i.e., the set E is the powerset 2^C of the set of query conditions C. The remaining steps, discussed in the next subsections, aim at computing the coefficient of each explanation. To illustrate the concepts introduced in the flow of the discussion, we rely on the scenario introduced in Example 4.2.2 (see Figure 4.12). Figure 4.13 illustrates several intermediate steps of *Ted++* that we will subsequently describe, proceeding in a bottom-up fashion. Note that the detailed presentation and discussion of the algorithm *Ted++* serves also for a proof sketch for Theorem 4.2.1.

So, let us discuss *Ted++* per step.

Algorithm 8: CoefficientEstimation

```
Input: $E$ explanations space, $\mathcal{P}$ valid partitioning of $\mathcal{S}$
for $\mathcal{E} \in E$ *access in ascending size order* do
    Compute $part_{\mathcal{E}}$;
    if $|\mathcal{E}| = 1$ then
        materialize $V_{\mathcal{E}}$;
        $\beta_{\mathcal{E}} \leftarrow$ Equation (B);
    else
        if $\alpha_{subcombination\ of\ \mathcal{E}} \neq 0$ then
            $\{\mathcal{E}1, \mathcal{E}2\} \leftarrow$ subCombinationsOf($\mathcal{E}$);
            $\Gamma_{12} \leftarrow \Gamma_1 \cap \Gamma_2$; *$\Gamma_i$ is the output schema of $V_{\mathcal{E}i}$*
            if $\Gamma_{12} \neq \emptyset$ then
                $V_{\mathcal{E}} \leftarrow V_{\mathcal{E}1} \bowtie_{\Gamma_{12}} V_{\mathcal{E}2}$;
                materialize $V_{\mathcal{E}}$;
            else
                $|V_{\mathcal{E}}| \leftarrow |V_{\mathcal{E}1}| * |V_{\mathcal{E}2}|$;
        else
            $|V_{\mathcal{E}}| \leftarrow |V_{\mathcal{E}1}| * |V_{\mathcal{E}2}|$;
        $\beta_{\mathcal{E}} \leftarrow \prod_{Part \in part_{\mathcal{E}}} |CT_{|Part}| - |(\bigcup_{i=1}^{n} V_{opi})^{ext}|$; * Equation (E) *
    $\alpha_{\mathcal{E}} \leftarrow$ Equation (A);
```

Partial Compatible Tuples Computation

Using the conditions in *WN*, *Ted++* partitions the schema $\mathcal{S}$ (Algorithm 7 line 2) into components of relations connected by the conditions in *WN*. This step is the same as the partitioning step in the *Ted* algorithm, relying on Lemma 2.2.1. It results into creating sets of *partial* compatible tuples, associated with the generated partitions. However, it should be clear that the partial compatible tuples are never combined through cross-product in *Ted++*, like it was the case in *Ted*.

Example 4.2.9. *The valid partitioning of $\mathcal{S}$ was found in Example 4.2.8 to be $Part_{RS}$=$\{R, S\}$ and $Part_T$=$\{T\}$. The sets of partial compatible tuples $CT_{|Part_{RS}}$ and $CT_{|Part_T}$ are given in the bottom line of Figure 4.13.*

Next, we compute the number of compatible tuples pruned out by each potential explanation, using the sets of partial compatible tuples. These numbers are used to calculate the coefficients of the explanations in the polynomial. From this point on we are only using compatible tuples, so we omit the word 'compatible' to lighten the discussion.

Polynomial Coefficient Estimation

Each set $\mathcal{E}$ in the powerset E is in fact a potential explanation that is further processed. This process is meant to associate with $\mathcal{E}$ (i) the set of partitions $part_{\mathcal{E}}$

over which $\mathcal{E}$ is defined, (ii) the view $V_{\mathcal{E}}$ of the passing partial tuples w.r.t. $\mathcal{E}$, and (iii) the number $\alpha_{\mathcal{E}}$ of tuples eliminated by $\mathcal{E}$. Note that we choose to compute passing rather than eliminated tuples as they are potentially less numerous. In an optimized version this decision could be made dynamically based on view cardinality estimation.

Algorithm 8 describes how we process E in ascending order of explanation size, i.e., from explanations containing one condition to the explanation containing them all. This enables us to reuse results obtained for sub-explanations and avoid cross product computations by mathematical calculations. Note that an explanation $\mathcal{E}'$ is a sub-explanation of $\mathcal{E}$ if $\mathcal{E}' \subseteq \mathcal{E}$.

We first determine the set of partitions associated with an explanation $\mathcal{E}$ as $part_{\mathcal{E}}=\{Part_{op} \mid op \in \mathcal{E}\}$, where $Part_{op}$ contains at least one relation over which op is specified.

Example 4.2.10. *Consider $\mathcal{E}_1=\{op_1\}$ and $\mathcal{E}_2=\{op_2\}$. From Figure 4.12(b) and the partitions in Figure 4.13, we can see that op_1 is specified over attributes of the relation R and S, so it is associated only with $Part_{RS}$. On the other side, op_2 is associated with $Part_{RS}$ and $Part_T$. Hence, $part_{\mathcal{E}_1}=\{Part_{RS}\}$ and $part_{\mathcal{E}_2}=\{Part_{RS}, Part_T\}$. Then, the explanation $\mathcal{E}=\{op_1, op_2\}$ is associated with the union of $part_{\mathcal{E}_1}$ and $part_{\mathcal{E}_2}$, resulting in $part_{\mathcal{E}}=\{Part_{RS}, Part_T\}$.*

We use Equation (A) to calculate the number $\alpha_{\mathcal{E}}$ of eliminated tuples, using the number $\beta_{\mathcal{E}}$ of eliminated *partial* tuples and the cardinality of the partitions not in $part_{\mathcal{E}}$. Intuitively, this formula 'fictionally' extends the partial tuples to "full" tuples over CT's schema.

$$\alpha_{\mathcal{E}} = \beta_{\mathcal{E}} * \prod_{Part \in \overline{part_{\mathcal{E}}}} |CT_{|Part}|, \tag{A}$$

where $\overline{part_{\mathcal{E}}}=\mathcal{P} \setminus part_{\mathcal{E}}$. Note that when $\overline{part_{\mathcal{E}}}$ is empty, we abusively consider that $\prod_{\emptyset}=1$.

The presentation now focuses on calculating $\beta_{\mathcal{E}}$, the only unknown value in Equation (A). Two cases arise depending on the size (a.k.a. number of conditions) of $\mathcal{E}$.

Atomic explanations. We start with atomic explanations $\mathcal{E}$ with only one condition op (Algorithm 8 lines 3-5). We firstly compute the set of *passing* partial tuples w.r.t. op, i.e., the tuples that satisfy op, and store them in the view V_{op}. Note that in this case (of atomic explanation) $part_{\mathcal{E}}$ is a set of at most 2 partitions.So,

$$V_{op} = \begin{cases} \pi_{\{R\_id \mid R \in Part\}}(\sigma_{op}[CT_{|Part}]) \text{ if } part_{\mathcal{E}}=\{Part\} \\ \pi_{\{R\_id \mid R \in Part_1 \cup Part_2\}}([CT_{|Part_1}] \bowtie_{op} [CT_{|Part_2}]) \\ \hfill \text{if } part_{\mathcal{E}}=\{Part_1, Part_2\} \end{cases}$$

The number of partial tuples eliminated by $\mathcal{E}$ is the number of all partial tuples in $part_{\mathcal{E}}$ (see Lemma 2.2.1) minus the passing ones.

$$\beta_{\mathcal{E}} \ = \prod_{Part \in part_{\mathcal{E}}} |CT_{|Part}| - |V_{op}| \tag{B}$$

Example 4.2.11. *For op_2, we have $part_{op_2}$=$\{Part_{RS}, Part_T\}$, so*

$$V_{op2}{=}\pi_{R\_Id,S\_Id,T\_Id}([CT_{|Part_{RS}}]\bowtie_{R.B=T.B}[CT_{|Part_T}])$$

This results in $|V_{op_2}|$=4, and by Equation (B) we obtain

$$\beta_{op2}{=}|CT_{|Part1}| * |CT_{|Part2}|{-}|V_{op2}|{=}3*4{-}4{=}8$$

Since all partitions of $\mathcal{P}$ are in $part_{op2}$, applying Equation (A) results in

$$\alpha_{op2}{=}\beta_{op2}{=}8$$

For op_3

$$\beta_{op3}{=}|CT_{|Part_T}|{-}V_{op3}{=}4{-}2{=}2$$

so

$$\alpha_{op3}{=}3*2=6$$

Figure 4.13 (second level) displays the results for all atomic explanations.

Non atomic explanations. Now, assume that $\mathcal{E}$=$\{op_1, \ldots, op_n\}$, $n > 1$ (Algorithm 8, lines 6-16). For the moment, we assume that the conditions in $\mathcal{E}$ share the same schema, so the intersection and union of V_{opi} for $i = 1, \ldots, n$ are well-defined. Firstly, we compute the view $V_{\mathcal{E}}$ of passing partial tuples w.r.t. $\mathcal{E}$ as $V_{\mathcal{E}}$=$V_{op1}\cap \ldots \cap V_{opn}$.

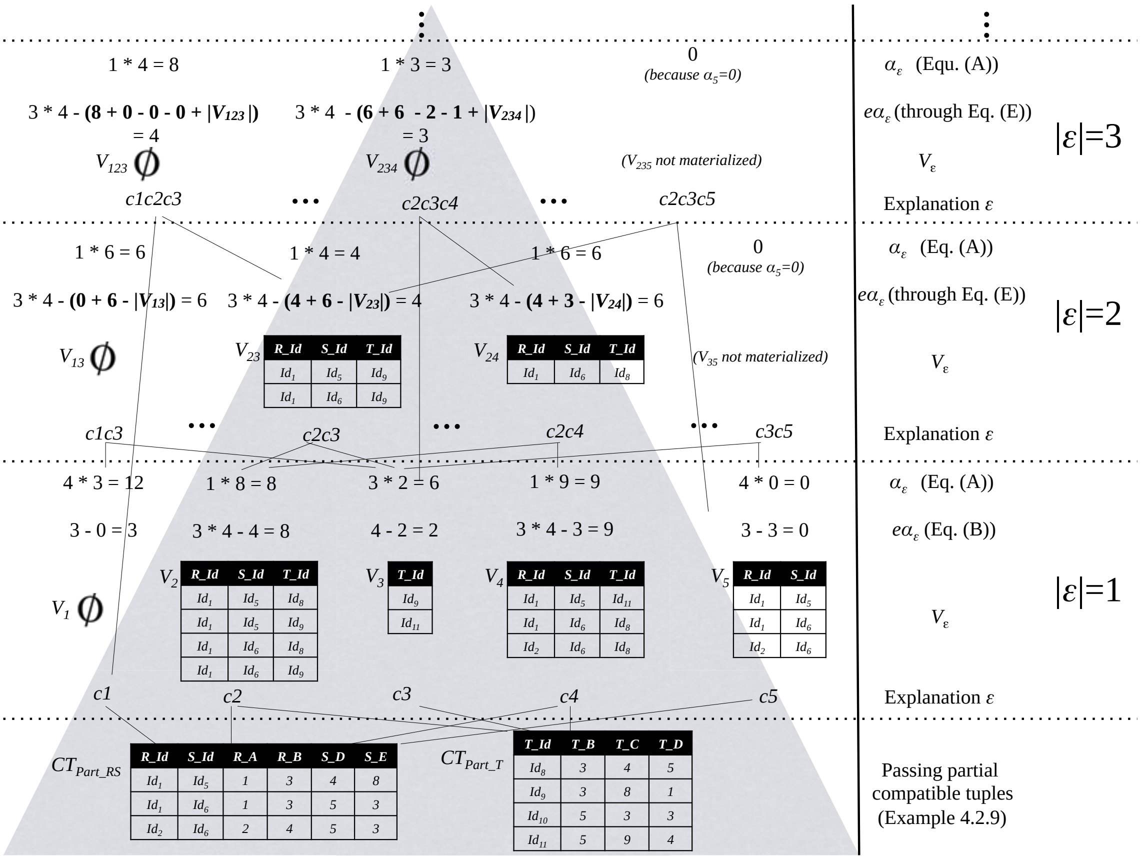
To compute the number of partial tuples pruned out by $\mathcal{E}$, we need to find the number of partial tuples pruned out by op_1 and ... and op_n, i.e., $\beta_{\mathcal{E}}$= $| \overline{V_{op1}} \cap \cdots \cap \overline{V_{opn}} |$. By the well-known *DeMorgan law* [Vau01], we have $\beta_{\mathcal{E}}$=$|\overline{V_{op1} \cup \cdots \cup V_{opn}}|$, which spares us from computing the complements of V_{opi}.

To compute the cardinality of the union of the V_{opi}, we rely on the *Principle of Inclusion and Exclusion for counting* [Hal98]:

$$|\bigcup_{i=1}^{n} V_{op_i}| = \sum_{\emptyset \neq J \subseteq [n]} (-1)^{|J|+1} \, |\bigcap_{j \in J} V_{opj}|$$

We further rewrite the previous formula to reuse results obtained for sub-combinations of $\mathcal{E}$, leading to Equation (C).

$$\begin{aligned} |\bigcup_{i=1}^{n} V_{opi}| &= |\bigcup_{i=1}^{n-1} V_{opi}| + |V_{opn}| \\ &\quad + \sum_{\emptyset \neq J \subseteq [n-1]} (-1)^{|J|} \, |\bigcap_{j \in J} V_{opj} \cap V_{opn}| \end{aligned} \tag{C}$$



V_{23}

R_Id	S_Id	T_Id
Id_1	Id_5	Id_9
Id_1	Id_6	Id_9

V_{24}

R_Id	S_Id	T_Id
Id_1	Id_6	Id_8

V_2

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_3

T_Id
Id_9
Id_{11}

V_4

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_5

R_Id	S_Id
Id_1	Id_5
Id_1	Id_6
Id_2	Id_6

CT_{Part_RS}

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

CT_{Part_T}

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 4.13: Running example with the different steps of *Ted++* (up to explanations of size 3) in Algorithm 7 and Algorithm 8.

At this point, we can compute $\beta_{\mathcal{E}}$ as all the variables of the problems have been rewritten in function of V_{op} views. However, so far we assumed that the conditions in $\mathcal{E}$ use the same schema. In the general case, this does not hold and we have to "extend" the schema of a view V_{op} to the one of $V_{\mathcal{E}}$, in order to ensure set operations to be well-defined. The cardinality of an extended V_{op}^{ext} is given by Equation (D).

$$| V_{op}^{ext} |= (\prod_{Part \in part_{\mathcal{E}} \setminus part_{op}} | CT_{|Part} |) * | V_{op} | \tag{D}$$

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

$$\begin{aligned} | (\bigcup_{i=1}^{n} V_{opi})^{ext} | = | (\bigcup_{i=1}^{n-1} V_{opi})^{ext} | + | V_{opn}^{ext} | \\ + \sum_{\emptyset \neq J \subseteq [n-1]} (-1)^{|J|} | (V_{(op_j)_{j \in J} op_n})^{ext} | \end{aligned} \tag{E}$$

The cardinalities of the views $V_{\mathcal{E}'} = V_{(op_j)_{j \in J} op_n}$ associated with $\mathcal{E}'$ for $| J |< n-1$, are sub-explanations of the explanation $\mathcal{E}$ and thus have already been computed by previous iterations. They only have to be extended to the schema of $V_{\mathcal{E}}$, if needed. When $|J|=n-1$, then $V_{\mathcal{E}'}=V_{\mathcal{E}}$. The view $V_{\mathcal{E}}$ is computed by the query $(part_{\mathcal{E}}, \{R\_Id | R \in part_{\mathcal{E}}\}, \{op \mid op \in \mathcal{E}\})$. However, there are cases when the computation of this query can be avoided as we stress out shortly below.

Now, we trivially compute the number $\beta_{\mathcal{E}}$ of eliminated partial tuples as the complement of $\mid (\bigcup_{i=1}^{n} V_{opi})^{ext} \mid$ (see Algorithm 8, line 17). The number of 'full' eliminated tuples is then calculated by Equation (A).

Example 4.2.12. *To illustrate the concepts introduced above, please follow the subsequent discussion in parallel with advising Figure 4.13. For brevity, we use subscript i instead of op_i, i.e., V_i means V_{op_i}.*

For the explanation op_2op_3, Equation (E) gives:

$$|(V_2 \cup V_3)^{ext}|=|V_2^{ext}| + |V_3^{ext}| - |(V_{23})^{ext}|$$

The schema of $part_{23}=\{Part_1, Part_2\}$ is $\{R\_Id, S\_Id, T\_Id\}$. The view V_2 has the same schema as the query Q, thus

$$|V_2^{ext}|=|V_2|=4$$

For V_3, the schema is $\{T\_Id\}$, so we in order to extend it to $\mathcal{S}_Q$ we apply Equation (D)

$$|V_3^{ext}|=|V_{Part_1}|* |V_3|=3*2=6$$

Now, we have to compute the cardinality $|V_{23}^{ext}|$. The conditions $R_B=T.B$ and $T.C\geq 8$ have one relation in common.So, the query statement $\{R,S,T\},\{R\_Id, S\_Id, T\_Id\}$ for V_{23} does not require cross product and is executed in order to compute that the size is $|V_{23} \models 2$ (as shown in Figure 4.13).

Finally, from Equation (E) we obtain

$$|(V_2 \cup V_3)^{ext}|=4+6-2=8$$

Since

$$|Part_{RS}| * |Part_T| = 12$$

then

$$\beta_{23}=12-8=4$$

and by Equation (A)

$$\alpha_{23}=4$$

We now focus on the explanation op_3op_5. The schemas of V_3 and V_5 are disjoint and intuitively $V_{35}=V_3 \times V_5$. As cross product is involved V_{35} is not computed; instead, we simply calculate

$$|V_{35}|=|V_3|*|V_5|=6$$

Then,

$$|\beta_{35}|=12-(12+6-6)=0$$

As we will see later, these steps are never performed in our algorithm. The fact that op_5 does not eliminate any tuple (see

$$\alpha_5{=}0$$

in Figure 4.13) implies that neither do any of its super-combinations. Thus, we know a priori that

$$\alpha_{35}{=}\alpha_{235}{=}\ldots{=}0$$

Finally, we illustrate the case of a bigger size combination, for example $op_2op_3op_4$ of size 3. Equation (E) yields

$$|(V_2 \cup V_3 \cup V_4)^{ext}|{=}|(V_2 \cup V_4)^{ext}| + |V_3^{ext}|{-}|(V_{23})^{ext}|{-}|(V_{43})^{ext}|{+}|(V_{243})^{ext}|$$

All terms of the right side of the equation are available from previous iterations, except for $|(V_{234})^{ext}|$. After executing the query for V_{234} we obtain So,

$$|(V_{234})^{ext}|{=}9$$

and

$$\beta_{234}{=}\alpha_{234}{=}12-9=3$$

In the same way, we compute all the possible explanations until we reach the explanation $op_1op_2op_3op_4op_5$.

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 $\mathcal{E}$. 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 $\mathcal{V}_\mathcal{E}$ through $\sim^*$.

When this partitioning is a singleton, $V_\mathcal{E}$ needs to be materialized (Algorithm 8, line 9). As previously mentioned, we materialize the view the query $V_\mathcal{E}$ using the query statement $(part_\mathcal{E}, \{R\_Id|R \in part_\mathcal{E}\}, \{op \mid op \in \mathcal{E}\})$. Nevertheless, we avoid the materialization of $V_\mathcal{E}$ if the partitioning is a singleton (Algorithm 8, line 9 & 16), when for some sub-combination $\mathcal{E}'$ of $\mathcal{E}$ it was computed that $\alpha_{\mathcal{E}'}{=}0$. In that case, we know a priori that $\alpha_\mathcal{E}{=}0$ (see Example 4.2.12).

If the partitioning is not a singleton, $V_\mathcal{E}$ is not materialized (Algorithm 8, line 14). For example, the partitioning for op_3op_5 is not a singleton and so we have $|V_{35}|{=}|V_3|{\times}|V_5|{=}6$.

Post-processing. In Algorithm 8 we associated with each possible explanation $\mathcal{E}$ the number of eliminated tuples $\alpha_\mathcal{E}$. However, $\alpha_\mathcal{E}$ includes any tuple eliminated by $\mathcal{E}$, even though the same tuples may be eliminated by some super-combinations of $\mathcal{E}$ (see Example 4.2.13). This means that for some tuples, more than one explanations have been assigned. To correct this, the last step of *Ted++* (Algorithm 7, line 6) calculates the coefficient for each explanation $\mathcal{E}$ by subtracting the coefficients of its super-explanations from $\alpha_\mathcal{E}$. This step is performed top-down. The coefficient of an explanation is thus

$$coef_{\mathcal{E}} = \alpha_{\mathcal{E}} - \left(\sum_{\mathcal{E} \subseteq \mathcal{E}'} coef_{\mathcal{E}'} \right) \tag{F}$$

Example 4.2.13. *Consider known $coef_{1234}$=2 and $coef_{123}$=2. We have found in Example 4.2.12 that α_{23}=4. With Equation (F), $coef_{23}$=4−2−2=0. In the same way $coef_{2}$=4−0−2−2=0. The algorithm leads to the expected Why-Not answer polynomial already provided in Example 4.2.4.*

Complexity Analysis In the pseudo-code provided by Algorithm 7, we can see that *Ted++* divides into the phases of (i) partitioning $\mathcal{S}$, (ii) materializing a view for each partition, (iii) computing the explanations, and (iv) computing the exact coefficients. When computing the explanations, according to Algorithm 8, *Ted++* iterates through $2^{|C|}$ condition combinations and for each, it decides upon view materialization (again through partitioning) before materializing it, or simply calculates $|V_{\mathcal{E}}|$ before applying equations to compute $\alpha_{\mathcal{E}}$. Overall, we consider that all mathematical computations are negligible so, the worst case complexities of steps (i) through (iv) are $O(|\mathcal{S}|+|WN|)+O(|\mathcal{S}|) + O(2^{|C|}(|\mathcal{S}| + |C|))+O(2^{|C|})$. For sufficiently large queries, we can assume that $|\mathcal{S}|+|C|<<2^{|C|}$, 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 $|\mathcal{I}_R|$ is the biggest size relation instance in $\mathcal{I}$. 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 $O(|\mathcal{I}_R|^{|\mathcal{S}|})$. This yields a combined complexity of $O(2^{|C|}|\mathcal{I}_R|^{|\mathcal{S}|})$. However, *Ted++* in the general case (more than one induced partitions), has a tighter upper bound: $O(|\mathcal{I}_R|^{k_{\mathcal{E}1}} + |\mathcal{I}_R|^{k_{\mathcal{E}2}} + \ldots + |\mathcal{I}_R|^{k_{\mathcal{E}N}})$, where $k_{\mathcal{E}}$ denotes the number of relations in the partitions in $part_{\mathcal{E}}$, for all combinations $\mathcal{E}$ and $N = 2^{|C|}$. It is easy to see that $|\mathcal{I}_R|^{k_{\mathcal{E}1}}+|\mathcal{I}_R|^{k_{\mathcal{E}2}}+\ldots+|\mathcal{I}_R|^{k_{\mathcal{E}N}} < 2^{|C|}|\mathcal{I}_R|^{|\mathcal{S}|}$, when there is more than one partition.

4.2.6 Experiments

This section presents an experimental evaluation of *Ted++* on real and synthetic datasets. In Section 4.2.6, we compare *Ted++* with the related algorithms returning query-based explanations (*Ted* [BHT14a], *NedExplain* and *Why-Not* [CJ09]). Section 4.2.6 studies the runtime of *Ted++* with respect to various parameters. We have implemented all algorithms in Java. 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 and use PostgreSQL 9.3 as database system.

Comparative Evaluation

The comparative evaluation of *Ted++* to *Why-Not* and *NedExplain* considers both efficiency (runtime) and effectiveness (explanation quality). When considering

Table 4.7: Queries for the scenarios in Table 4.8

Query	**Expression**
Q1	$\pi_{P.name,C.type}[C \bowtie_{sector} W \bowtie_{witnessName} S \bowtie_{hair,clothes} P]$
Q2	$\pi_{P.name,C.type}[\sigma_{C.sector>99}[C] \bowtie_{sector} W \bowtie_{witnessName} S \bowtie_{hair,clothes} P]$
Q3	$\pi_{W.name,C2.type}[W \bowtie_{sector2} C2 \bowtie_{sector1} \sigma_{C.type=Aiding}[C]]$
Q4	$\pi_{P2.name}[P2 \bowtie_{!name,hair} \sigma_{P1.name<B}[P1]]$
Q5	$\pi_{name,L.locationid}[L \bowtie_{movieId} \sigma_{M.year>2009}[M] \bowtie_{name} \sigma_{R.rating\geq 8}[R]]$
Q6	$\pi_{Co.firstname,Co.lastname}[\sigma_{AA.party=Republican}[AA] \bowtie_{id} \sigma_{Co.Byear>1970}[Co]]$
Q7	$\pi_{SPO.sponsorId,SPO.sponsorln,E.camount}[E \bowtie_{eId} \sigma_{ES.sub=Sen.\ Com.}[ES] \bowtie_{id} \sigma_{SPO.party=Rep.}[SPO]]$
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$_{tpch}$	$C \bowtie_{ckey} \sigma_{odate<1998-07-21}[O] \bowtie_{okey} \sigma_{sdate>1998-07-21}[L]$

efficiency, we also include *Ted* in the discussion (*Ted* producing the same Why-Not answer as *Ted++*).

For the experiments we have used data from three databases named *crime*, *imdb*, and *gov* (the same as in Section 4.1.6). For each dataset, we have created a series of scenarios (crime1 to gov5 in Table 4.8 - ignore remaining scenarios for the moment). Each scenario consists of a query further defined in Table 4.7 (Q1 to Q7) and a simple Why-Not question, as *Why-Not* and *NedExplain* support only this type of Why-Not question. We have designed queries with a small set of conditions (Q6) and others with more conditions (Q1, Q3, Q5, Q7), containing self-joins (Q3, Q4), having empty intermediate results (Q2), as well as containing inequalities (Q2, Q4, Q5, Q6).

Table 4.8: 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>=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}

Why-Not Answer Evaluation

In Table 3.1 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 answer to sets of conditions, e.g., $3op_3 * op_4 + 2op_3 * op_6$ maps to $\{\{op_3, op_4\}, \{op_3, op_6\}\}$. For conciseness, we abbreviate condition sets, e.g., the previous set of explanations is written $\{op_{34}, op_{36}\}$.

Table 4.9 summarizes the Why-Not answers of the three algorithms. These scenarios show that the explanations provided by *NedExplain* or *Why-Not* are incomplete, in two ways. 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 provided may lack some conditions, which can drive the user to fruitless fixing attempts. On the contrary, *Ted++* produces all the possible, complete explanations.

For the first claim, consider the scenario $gov2$. *Why-Not* returns the explanation op_1, while *NedExplain* returns op_3. Indeed, both the explanations op_1 and op_3 are picky. However, if a developer was relying on the results of one of the two algorithms, she would miss the alternative existing explanations. On the contrary, *Ted++* computes both these alternatives. Then, consider $crime8$. *NedExplain* returns the join op_2 ($S \bowtie_{hair} P$) - *Why-Not* does not produce any explanations. *Ted++* indicates that except for this join, the selection op_3 ($\sigma_{name<`B'}[P]$) 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 claim consider $crime5$. *NedExplain* returns op_1 ($C \bowtie_{sector} W$). The Why-Not answer polynomial of *Ted++* does not contain the atomic explanation op_1, but some combinations include op_1, like op_{15}. Because the explanation by *NedExplain* is incomplete, a repairing attempt of op_1 alone will never yield the desired results. Similarly, $crime7$ illustrates a case, where the *Why-Not* algorithm produces an explanation (op_3) that misses some conditions. Then, in gov3 *NedExplain* and *Why-Not* both return op_2. However, let us now assume the developer prefers to not change this condition. Keeping in mind that the answers of these algorithms may change when changing the query tree, she may start trying different trees to possibly obtain a Why-Not answer without op_2. The explanation of *Ted++* prevents her from spending any effort on this 'quest', as it shows that all explanations contain op_2.

Having mapped the Why-Not answer polynomial of *Ted++* to a set of explanations, the usefulness of the coefficients of the polynomial has been neglected. For example, the Why-Not answer polynomial of $crime8$ is $2384 * op_{23} + 20 * op_3 + 4 * op_1 + 8 * op_2$. Assume that the developer would like to recover at least 5 missing tuples, by changing as few conditions as possible. The polynomial implies to change either op_3 or op_2: they both require one condition change and provide the possibility of obtaining up to 20 and 8 missing tuples, respectively. op_1 can recover up to 4 tuples, whereas $op_2 op_3$ require two condition changes. Clearly, the results of *NedExplain* or *Why-Not* are not informative enough to allow for such a decision.

Table 4.9: *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

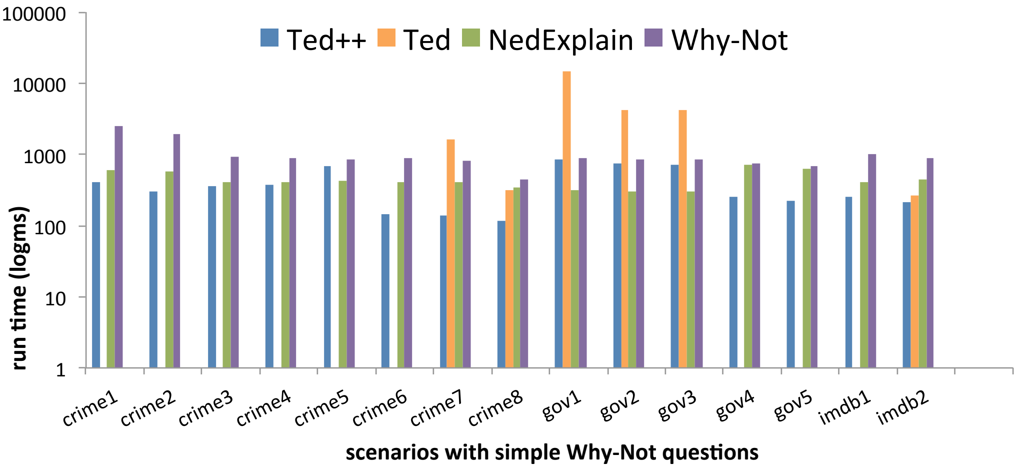


Figure 4.14: Runtimes for *Ted++*, *Ted*, *NedExplain* and *Why-Not*

Runtime Evaluation *Ted++* vs. *NedExplain* and *Why-Not*. Figure 4.14 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 lineage tables stored in Trio. As such, for each intermediate result a table is maintained with the lineage of the tuples, which is afterwards queried to find if successors of the compatible tuples exist in the intermediate results. As already stated in [CJ09], this design choice slows down

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

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., $gov1$).

Ted++* vs. *Ted. Fig. 4.14 reports runtimes for *Ted* on 6 out of 15 scenarios as for the others, *Ted* runs out of time. To examine this behaviour, we compare the time distribution in *Ted* and *Ted++*, as in Fig. 4.15. The algorithms are divided in four common phases. Note that, for scenarios $crime7$, $gov1 - gov3$ the execution time for *Ted* is much higher compared to the other scenarios and to the runtime of *Ted++*; in this cases the diagram for *Ted* is not displayed in total (the runtime of the coefficientEstimation phase is seen as label on the respective column).

As said in Section 4.2.5, *Ted*'s main issue w.r.t. efficiency is its strong dependence on the number of compatible tuples. This is experimentally observed in Figure 4.15: with the growth of the set of compatible tuples in the scenarios, the time dedicated to coefficientEstimation also increases (the scenarios are reported in an ascending order of number of compatible tuples). *Ted++* depends on the number of compatible tuples as well, but in a less prominent manner than *Ted.* This can be seen in $crime8$, $crime7$, $gov3$, and $gov1$; while the number of tuples grows, *Ted++*'s runtime remains roughly steady.

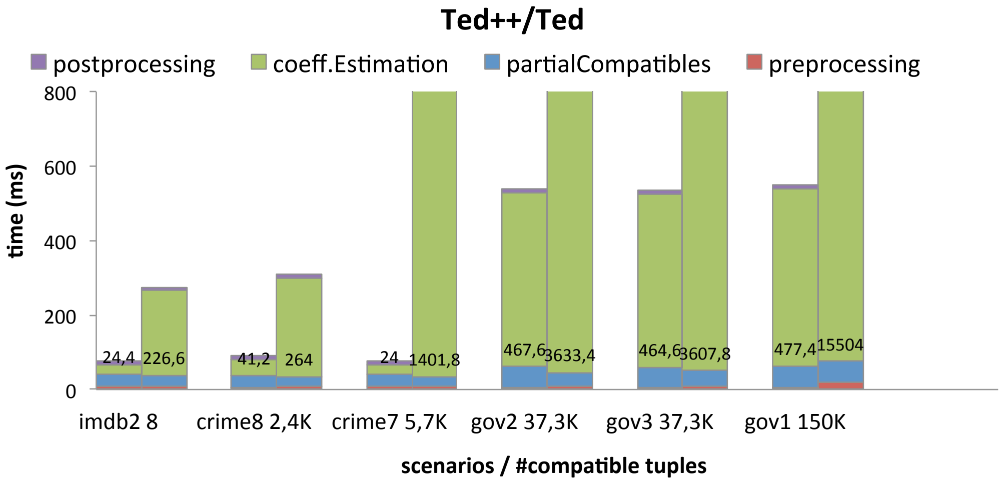
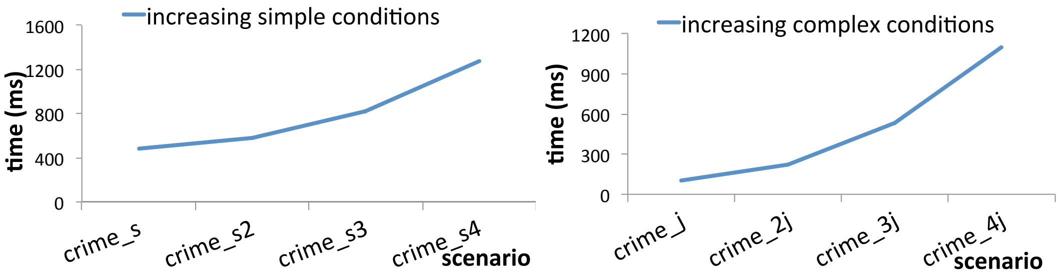


Figure 4.15: *Ted++* and *Ted* runtime distribution



(a) simple conditions (b) complex conditions

Figure 4.16: *Ted++* runtime w.r.t. number of conditions in Q

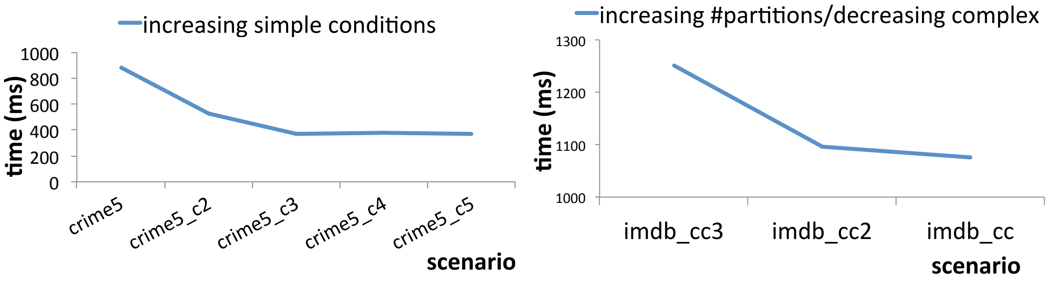
Ted++ Analysis

We now study *Ted++*'s runtime w.r.t. the following parameters: (i) the type (simple or complex) of the input query Q and the size of its condition set C_Q, (ii) the type of the Why-Not question (simple or complex) and the number and selectivity of the conditions in Why-Not question, and (iii) the size of the database instance $\mathcal{I}$. 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.

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/). We have generated instances of 1GB and 10GB and further produced smaller data sets of 10MB and 100MB to obtain a series of datasets whose size differs by a factor of 10. In this paper, we report results for the original query Q3 of the TPC-H set of queries. It includes two 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 Q$_s$ to Q$_{tpch}$ (Table 4.7) and the scenarios are *crime$_s$-tpch$_c$* (Table 4.8).

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, Q$_{s3}$, Q$_{s4}$. The evolution of *Ted++* runtime for these queries is shown in Figure 4.16 (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 Figure 4.16 (b).

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



(a) simple conditions

(b) complex conditions

Figure 4.17: *Ted++* runtime w.r.t. number of conditions in *WN*

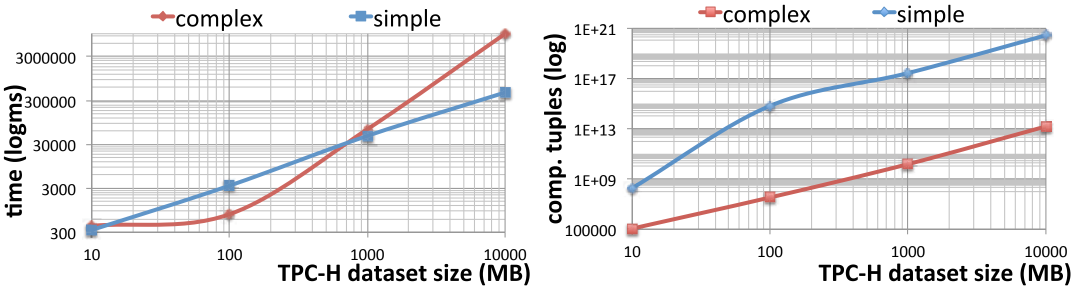
eliminated tuples materialization.

Adjusting the Why-Not question.
The scenarios considered for Figure 4.17 (a) have as starting point the simple Why-Not question of *crime*5 (see Table 4.8). Then, keeping the same input instance and query, we add attribute-constant comparisons (i.e., simple conditions) to the Why-Not question, a procedure resulting in fewer compatible tuples in each step. As expected, the more conditions (the less tuples) the faster the Why-Not answer is returned, until we reach a certain point (here from $crime5_{c3}$ on). From this point, the runtime is dominated by the time to communicate with the database that is constant for all scenarios.

In Figure 4.17 (b) we examine complex Why-Not questions. As we add complex conditions in a Why-Not question, the number of generated partitions (potentially) drops as more relations are included in a same partition. To study the impact of the induced number of partitions in isolation, we *keep the number of the compatible tuples constant* in our series of complex scenarios ($imdb_{cc}$-$imdb_{cc3}$). The number of partitions entailed by $imdb_{cc}$, $imdb_{cc2}$, and $imdb_{cc3}$ are 3, 2, and 1, respectively. The results of Figure 4.17 (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. For the last set of experiments we increase the database size, for scenarios with one simple or one complex Why-Not question *WN*, for the same query Q_{tpch}. 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.

Figure 4.18 (a) shows the runtime for both scenarios. The increasing runtime is tightly coupled to the fact that the number of computed tuples is rising proportionally to the database size, as shown in Figure 4.18 (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,



(a) runtime

(b) # compatible tuples

Figure 4.18: *Ted++* (a) runtime, and (b) number of compatible tuples for increasing database size, complex and simple *WN*

runtime deteriorates in a steady pace. This behavior is aligned with the theoretical study; when the number of partitions is decreasing the complexity rises.

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

4.2.7 Theoretical discussion

In this section we delve into a more theoretical discussion about interesting properties of the Why-Not answer polynomials. First, we show how Why-Not answer polynomials fit in different relational data models with bag and probabilistic semantics and that as in provenance semirings [GKT07] our Why-Not answer polynomials can be used as a unified framework to capture query based explanations in these settings. Second, we focus on equivalent queries and the relationship that is established among the Why-Not answer polynomials associated with each, given a Why-Not question. By showing that isomorphic queries yield equivalent Why-Not answer polynomials, we are able to prove that our method is invariant to query tree variations resulting from operator reordering and that the explanations produced by *NedExplain* are subsumed by the Why-Not answer polynomial. In this way, we show that Why-Not answer polynomials are complete and correct query based explanations in comparison with the other approaches computing Why-Not answers. Third, we show that equivalent queries in general are not equivalent w.r.t. Why-Not answer polynomials, by minimized tableaux queries. We further define an approximation for the Why-Not answer polynomial PEX for a query Q, obtained by considering a minimized (through tableau minimization) version of Q. We argue that in settings where a much smaller query (in number of joins) can result from tableau minimization, this minimized query can be more efficiently used to obtain an approximation of the Why-Not answer polynomial of the original query.

Why-Not answer polynomial for databases under bag and probabilistic semantics

So far, we have considered databases under *set* semantics only. In this section, we discuss how the definition of the Why-Not answer polynomial (Definition 4.2.2) extends to settings with conjunctive queries over bag and probabilistic semantics databases.

K-relations, as described in [GKT07], capture in a unified manner relations under set, bag, and probabilistic semantics. Briefly, tuples in a K-relation are annotated with elements from a set K. In our case, we consider that K is a set of unique tuple identifiers, similar to our special attribute R_Id (see Section 2.1).

We use the notion of *how*-provenance of tuples in the result of a query Q. The how-provenance of $t{\in}Q(\mathcal{I})$ is the polynomial obtained by the positive algebra RA^+ on K-relations, proposed in [GKT07]. Briefly, each t is annotated with a polynomial where variables are tuple identifiers and coefficients are natural numbers. Roughly, if t results from a selection operator on t_1 annotated with Id_1, then t is also annotated with Id_1. If t is the result of the join of t_1 and t_2, then t is annotated with $Id_1 Id_2$. If t results from t_1 annotated with Id_1 or t_2 annotated with Id_2 passing through a union operator, then t is annotated with $Id_1 + Id_2$. In general, the how-provenance of an RA^+ query result is the semiring of polynomials with integer coefficients and variables from the commutative semiring $(K, +, \bullet, 0, 1)$). More details on semirings and how-provenance are provided in Section 3.1.

We compute the *generalized* Why-Not answer polynomial PEX_{gen} as follows. Firstly, we compute the how-provenance for compatible tuples in CT by evaluation of the query Q_{WN} (Definition 2.2.5) w.r.t. the algebra in [GKT07]. Recall that Q_{WN} contains only selection and join operators. Thus, each compatible tuple τ in CT is annotated with its how-provenance polynomial consisting only of one term (since there is no union), denoted by η_τ.

Example 4.2.14. *Consider again Example 4.2.2. The relations R, S, T can be considered as K-relations, whose annotation attribute (e.g., $\mathcal{R}\_Id$) values are elements of the set $K = \{Id_1, \ldots, Id11\}$. Consider now the compatible tuple*

$$\tau_1{=}(R\_Id{:}Id_1, R.A{:}1, R.B{:}3, S\_Id{:}Id_5, S.D{:}4, S.E{:}8, T\_Id{:}Id_8, T.B{:}3, T.C{:}4, T.D{:}5)$$

This tuple results from the query

$$Q_{WN}{=}\sigma_{R.B<S.D,T.C\leq 9}[R \times S \times T]$$

and is annotated with the polynomial

$$\eta_{\tau_1}{=}Id_1 Id_5 Id_8$$

from the commutative semiring $(K, +, \bullet, 0, 1)$. So, η_{τ_1} corresponds to the how-provenance of τ_1.

Then, we combine the expressions of *how* and *why-not* provenance. In order to do this, for each compatible tuple τ in CT, we combine its how-provenance polynomial η_τ with its explanation $\mathcal{E}_\tau$ (Definition 4.2.1). So, each τ is associated with the expression $\eta_\tau\mathcal{E}_\tau$. Note that even though $\mathcal{E}_\tau$ is defined as a set of conditions, here we use it also for the term resulting from the conditions in $\mathcal{E}_\tau$.

Example 4.2.15. *In Example 4.2.3 the explanation for τ_1 was found to be $\mathcal{E}_{\tau_1}$=$\{op_1op_3op_4\}$. Now, combining $\mathcal{E}_{\tau_1}$ and η_τ we obtain the expression*

$$\eta_\tau\mathcal{E}_{\tau_1}{=}Id_1Id_5Id_8op_1op_3op_4$$

Finally, we sum the combined expressions for all compatible tuples, which leads to the expression $\sum\limits_{\tau\in CT}\eta_\tau\mathcal{E}_\tau$.

We now comment on how PEX_{gen} is instantiated to deal either with the set, bag, or probabilistic semantics. Indeed, the 'specialization' of PEX_{gen} relies on the interpretation of the elements in K, that is on a function $Eval$ from K (the set of annotations or tuple identifiers) to some set L containing values for the tuple identifiers. Intuitively, the values in L relate to the semantics of the given database model, i.e., how many duplicates of one tuple exist in a relation or what is the probability of one tuple to occur in a relation.

For the set semantics each tuple in a relation occurs only once, which means that L is the singleton $\{1\}$. Thus every tuple identifier is mapped to 1. It is then quite obvious to note that $PEX_{gen} = PEX$ (Definition 4.2.2) for set semantics because for every τ, $\eta_\tau = 1$.

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 occurences. 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 answer is parametrized by the mapping $Eval$ of the annotations (elements in K) in the set L. Note that $Eval$ is naturally extended to cover the compositional expressions η as $Eval(\eta) = Eval(Id_1 \dots Id_n) = Eval(Id_1)\dots Eval(Id_n)$, where n is the number of the involved annotations in η.

Definition 4.2.5. *(Generalized Why-Not explanation polynomial) Given a query Q over a database schema $\mathcal{S}$ of K-relations, the generalized Why-Not answer polynomial for WN is*

$$\mathrm{PEX}_{gen} = \sum_{\mathcal{E}\in E}\left(\sum_{\tau\in CT s.t.\ \mathcal{E}_\tau=\mathcal{E}} Eval(\eta_\tau)\right)\mathcal{E}$$

where E=2^C, η_τ is the how-provenance of $\tau \in CT$, and $Eval$:$K \to L$ maps the elements of K to values in L.

Example 4.2.16. *Let us continue Example 4.2.15 and firstly consider bag semantics. Assume that $Eval(Id_1) = 2, Eval(Id_5) = 1, Eval(Id_8) = 3$. Then, for τ_1 we*

Table 4.10: Mapping functions

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

obtain the expression $2*1*3op_1op_3op_4 = 6op_1op_3op_4$. This means (for convenience we consider that τ_1 is the only compatible tuple here) that up to 6 missing tuples can be retrieved if $op_1op_3op_4$ is changed.

If we consider the probabilistic semantics let us assume that $Eval(Id_1) = 0.1$, $Eval(Id_5) = 0.5$, and $Eval(Id_8) = 0.5$. Then, for τ_1 we obtain the expression $0.025op_1op_3op_4$. Here the coefficient provides the upper bound of the probability with which a missing tuple will be retrieved if $op_1op_3op_4$ is fixed.

Why-Not answer polynomial and equivalent queries

In this section, we discuss some properties of the Ted Why-Not answer (*PEX*) w.r.t. equivalent queries. First, we show that *PEX* is robust for isomorphic queries. This provides the basis for showing that *PEX* subsumes the Why-Not answer by *NedExplain* (Section 4.1). Second, we show that the robustness of *PEX* does not hold for equivalent conjunctive queries in general. Moreover, we investigate the behaviour of *PEX* for equivalent queries Q and Q' s.t. Q' is obtained by tableau minimization of Q. This leads us to consider an approximate Ted Why-Not answer for Q when the Ted Why-Not answer for Q' is available.

The discussion requires expressing a conjunctive query (under set semantics) as a query tableau Q (Definition 2.1.8). For this reason, first we show how a Why-Not question and a Why-Not answer polynomial are defined for tableaux queries. Moreover, a couple of complementary functions are defined and illustrated in Table 4.10. The function h_{Att} is used to map schema attributes to variables of the tableau T_Q. The function $full$ is used to map a tuple identifier to its full database schema, and thus obtain all the involved attributes.

Function h_{Att} is extended to tableaux and to sets of conditions, whereas the function $full$ naturally extends to compatible tuples, e.g.,
$full(Id_1Id_5)$=$(R.A{:}1, R.B{:}3, S.C{:}1, S.D{:}4, S.E{:}8)$.

A Why-Not question vWN w.r.t. Q is expressed as a set of conditions over the variables in the summary of the query tableau Q. Thus, given the Why-Not question WN expressed over attributes it holds that vWN=$h_{Att}(WN)$.

Example 4.2.17. *Given the scenario of Example 4.2.2, the Why-Not question is expressed by vWN=$\{x_2<x_4, x_7\leq 9\}$.*

We further introduce the query tableau T_{vWN}=$(var(T_Q), T_Q, WN)$ to model the query Q_{WN} (Definition 2.2.5, page 20). Table 4.11 displays the tableau T_{vWN} for our running example.

Table 4.11: Tableau T_{vWN} corresponding to Q_{WN}

	R.A	R.B	S.C	S.D	S.E	T.B	T.C	T.D	*vWN*
R	x_1	x_2							$x_2 < x_4$
S			x_3	x_4	x_5				
T						x_6	x_7	x_8	$x_7 \leq 9$

Table 4.12: T_{τ_1}

	R.A	R.B	S.C	S.D	S.E	T.B	T.C	T.D	$cond_\tau$	C
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	$x_6{=}3, x_7{=}4, x_8{=}5$	$x_7 \geq 8$

Example 4.2.18. *Table 4.11 shows the tableau representation of WN for our running example. The valuation of this tableau over the instance in Figure 4.12(a) provides the set of compatible tuples CT.*

The next step is to find the Why-Not answer polynomial. We start with the explanation for one compatible tuple τ.

To find the query conditions that prune τ from the query result, we use the tableau T_τ for τ. More precisely, T_τ has the same body as the query tableau Q, but except for the query conditions C it incorporates the conditions $cond_\tau$ imposed over the attributes of the database schema by the tuple τ. Moreover, it has no summary row because the purpose of this tableau is not to model a query. Intuitively, we need to find which conditions in the column C contradict conditions in the column $cond_\tau$.

Definition 4.2.6 (Table T_τ)**.** *Given a compatible tuple τ w.r.t. a Why-Not question vWN and a query Q, the tableau T_τ associated with τ is defined by $(\_, T_Q, cond_\tau \cup C)$, where $cond_\tau$ is the set of conditions over $var(T_Q)$ induced by $full(\tau)$.*

T_τ is used to compute the explanation $\mathcal{E}_\tau$ by identifying the *picky* conditions in the query Q (elements of C). Note that the explanation $\mathcal{E}_\tau$ is formulated using variables rather than attributes. However, we can translate these explanations back to an attributes expression using the function h_{att}. Finally, having all the explanations for the compatible tuples, the Why-Not answer polynomial *PEX* is defined as in Definition 4.2.2, page 82.

Example 4.2.19. *Consider the compatible tuple τ_1 and the tableau depicted in Table 4.12. The condition $x_1{=}1$ in $cond_\tau$ contradicts the condition $x_1{>}3$ of C. This leads us to conclude that $x_1{>}3$ is a picky condition. The conditions involving x_2 in $cond_\tau$ and C 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 and eventually obtain the explanation w.r.t. τ_1 that is $\{x_1{>}3, x_7{\geq}8, x_4{=}x_8\}$.

Let us now continue the discussion for the equivalent queries.

R

A	B	C
a_3	b_1	c_1
a_1	b_2	c_2
a_1	b_1	c_3
a_2	b_1	c_2

S

A	B
a_1	b_2
a_1	b_3
a_2	b_1

(a) Database instance

	$R_1.A$	$R_1.B$	$R_1.C$	S.A	S.B	$R_2.A$	$R_2.B$	$R_2.C$	C_Q
R_1	x_1	x_2	x_3						$x_1 = a_1 \wedge x_2 = x_5$
S				x_4	x_5				$x_2 = x_5 \wedge x_7 = x_5$
R_2						x_6	x_7	x_8	$x_6 = a_1 \wedge x_7 = x_5$
s_Q			x_3	x_4					

(b) Query Q

	$R_1.A$	$R_1.B$	$R_1.C$	S.A	S.B	$R_2.A$	$R_2.B$	$R_2.C$	$C_{Q'}$
R_1	x'_1	x'_2	x'_3						$x'_1 = a_1 \wedge x'_2 = x'_5$
S				x'_4	x'_5				$x'_2 = x'_5 \wedge x'_7 = x'_5$
R_2						x'_6	x'_7	x'_8	$x'_6 = a_1 \wedge x'_7 = x'_5$
$s_{Q'}$			x'_3	x'_4					

(c) Isomorphic query Q'

name	op_1	op_2	op_3	op_4
condition	x_1=a_1	x_6=a_1	x_2=x_5	x_5=x_7

(d) Naming conditions of Q

Figure 4.19: Database instance and isomorphic queries

Isomorphic queries

We start by considering the class of equivalent isomorphic queries. Intuitively, two queries are isomorphic if they are the same up to variable renaming.

Definition 4.2.7. *(Isomorphic queries) Let Q=(s_Q, T_Q, C_Q) and Q'=$(s_{Q'}, T_{Q'}, C_{Q'})$ be two queries. Assume the isomorphism ϱ:$var(Q) \to var(Q')$. Then, Q and Q' are isomorphic through ϱ iff*

1. *every row R of Q is mapped to a row R' of Q', s.t. $T_{Q'}(R')$=$\varrho(T_Q(R))$,*
2. *$s_{Q'}$=$\varrho(s_Q)$,*
3. *$C_{Q'}$=$\{op' \mid op'$=$\varrho(op)$ and $op \in C_Q\}$, and*
4. *Q and Q' have the same number of rows.*

Example 4.2.20. *Figure 4.19 (a) and (b) display a sample database instance and a query Q over two relations R and S. The result of the query is $Q(\mathcal{I}) = \{c_3a_2, c_2a_1\}$. Obviously, the query Q' displayed in Figure 4.19 (c) is isomorphic to Q, since the variable x_i in Q is mapped to x'_i in Q' for $i = 1, \ldots, 8$ and the constant a_1 is mapped to itself.*

Obviously, two isomorphic queries Q and Q' are equivalent w.r.t. *PEX*. This means that for any Why-Not question and any instance $\mathcal{I}$, the *PEX* for Q and the *PEX* for Q' are the same (up to variable renaming).

Lemma 4.2.1. *(Equivalent queries w.r.t.* PEX*)*

Let Q=(s_Q, T_Q, C_Q) and Q'=$(s_{Q'}, T_{Q'}, C_{Q'})$ be two isomorphic queries and ϱ be the isomorphism s.t. $Q' = \varrho(Q)$. Then for any Why-Not question vWN w.r.t. Q and $\mathcal{I}$ over $\mathcal{S}_Q$, it holds that

$$\text{PEX}(Q', vWN', \mathcal{I}) = \varrho(\text{PEX}(Q, vWN, \mathcal{I})), \ \textit{where } vWN' = \varrho(vWN)$$

Example 4.2.21. *Continuing Example 4.2.20 consider now the Why-Not question $vWN{=}(x_3{=}c_2, x_4{=}a_2)$. For convenience, the conditions in C are named as in Figure 4.19 (d). The Why-Not answer polynomial for Q is*

$$\text{PEX}{=}\text{PEX}(Q, vWN, \mathcal{I}){=}op_3 + 2op_2op_3 + op_3op_4 + op_1 + 2op_1op_2 + op_1op_4$$

For Q', using the isomorphism relation that maps x to x', it holds that the Why-Not question is

$$vWN'{=}\varrho(vWN){=}(x_3'{=}c_2, x_4'{=}a_2)$$

If we assign to each condition $op'{=}\varrho(op)$ the same name as for op (where $op{\in}C_Q, op'{\in}C_{Q'}$) we obtain the same Why-Not answer polynomial, that is

$$\text{PEX}'{=}\varrho(\text{PEX}){=}op_3 + 2op_2op_3 + op_3op_4 + op_1 + 2op_1op_2 + op_1op_4$$

Now, we examine the relationship between the Why-Not answer polynomial returned by Ted (*PEX*) and the condensed Why-Not answer returned by *NedExplain* (NEX). For the sake of the discussion, we partially reproduce here the Why-Not answer definition for *NedExplain.* Recall that *NedExplain* considers queries as query trees. To simplify the discussion, we assume that query trees are built using (i) relation schemas as leaf nodes, and (ii) cross product × and selection σ_c, where *op* is a condition, as internal nodes. Without loss of generality, we do not consider projection here.

Definition 4.2.8. *(*NedExplain *Why-Not answer)*

Let $\mathcal{T}$ be a query tree and WN be a Why-Not question, and let CT be the set of compatible tuples. Then, the NedExplain *Why-Not answer for WN w.r.t. $\mathcal{T}$ and $\mathcal{I}$, denoted $NEX(\mathcal{T}, WN, \mathcal{I})$, is defined as the set of picky subtrees of $\mathcal{T}$ (a.k.a. subqueries) w.r.t. some compatible tuple of CT.*

In order to compare *PEX* with NEX, we reformulate *PEX* for query trees as well. To this end, we associate tableau queries with query trees and vice versa.

We start by associating a query tree $\mathcal{T}$ to a query tableau Q. Note that this tree is not unique, however, for a given query Q, all such trees are isomorphic. Moreover, isomorphic queries have isomorphic trees.

Definition 4.2.9 (Query tree for Q)**.** *Let $Q{=}(\_, T_Q, C)$ be a query tableau and assume that $|\mathcal{S}_Q|{=}n$. The set opSet of operators associated with Q is the set of selections $\{\sigma_{h^{-1}_{Att}(op)} | op{\in}C\}$.*

A query tree $\mathcal{T}$ is a tree for Q iff

1. *it has exactly $n-1$ cross product nodes,*
2. *it has exactly one node for each operator in opSet,*
3. *it has exactly one leaf node for each relation R in $\mathcal{S}_Q$, and*
4. *$\mathcal{T}$ is syntactically well-formed (i.e., the input schema of a condition complies with the target schema of the node).*

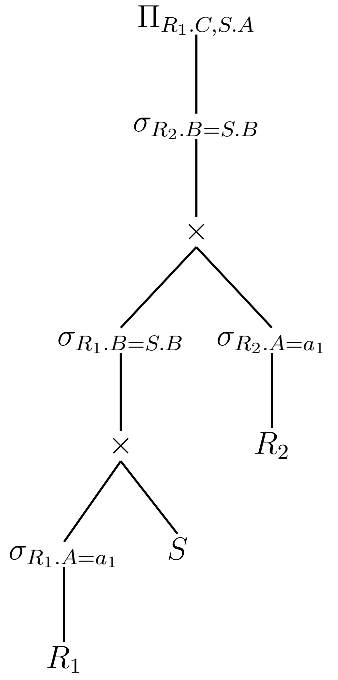


Figure 4.20: A query tree $\mathcal{T}$ for Q

Example 4.2.22. *Query Q (Figure 4.19 (b)) can be transformed to the query tree $\mathcal{T}$ in Figure 4.20.*

In the other direction, we associate a query Q to a tree $\mathcal{T}$. Note that a query tree is associated with exactly one query tableau and that trees resulting from nodes reordering, are associated with the same tableau. Moreover, isomorphic trees (trees obtained by variable renaming and node reordering) are associated with isomorphic tableaux.

To clarify, nodes reordering corresponds to moving query operators up or down on the query tree, preserving the semantics of the query (thus using the well known rewriting rules for relational algebra).

Definition 4.2.10 (Query tableau for $\mathcal{T}$). *Let $\mathcal{T}$ be a query tree with n leaf nodes, $n{-}1$ cross product nodes and let OpSet be the set of conditions associated with the non-cross product nodes (selection nodes) in $\mathcal{T}$. The query tableau $Q{=}(\_, T_Q, C)$ is the tableau for $\mathcal{T}$ iff*

1. *it has exactly one row for each leaf node of $\mathcal{T}$,*
2. *for each row R, $T_Q(R) = h_{Att}(\mathcal{A}(R))$, and*
3. *$C{=}h_{Att}(OpSet)$.*

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 (or isomorphic) tableau representation have the same *PEX*. Based on this statement and Lemma 4.2.1, we formally define a class of query trees equivalent w.r.t. *PEX*, also illustrated in Figure 4.21.

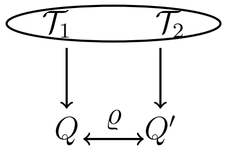


Figure 4.21: Equivalent query trees w.r.t. *PEX*

Lemma 4.2.2. *(Equivalent query trees w.r.t.* PEX*)*
Let $\mathcal{T}$ be a query tree for Q and $\mathcal{T}'$ be a query tree for Q'. If Q and Q' are isomorphic tableau queries, then $\mathcal{T}$ and $\mathcal{T}'$ are equivalent query trees w.r.t. PEX.

The above lemma states that reordered query trees have the same *PEX*. This is a stepping stone to the next theorem, stating that *PEX* subsumes NEX.

Theorem 4.2.2. *(*PEX $-$ NEX *subsumption)*
Let Q be a query tableau over the schema $\mathcal{S}_Q$ and $\mathcal{I}$ be an instance over $\mathcal{S}_Q$. Let vWN be a simple Why-Not question for the query tableau Q. Assume that $\mathcal{T}$ is a query tree for Q.

Let NEX=$NEX(\mathcal{T}, WN, \mathcal{I})$ and PEX=PEX$(Q, vWN, \mathcal{I})$ *be the Why-Not answer by* NedExplain *and* Ted *respectively. Then, if op is the condition of the root operator of a subquery $\mathcal{T}'$ and x denotes a term of* PEX*, it holds that*

$$\forall \mathcal{T}' \in NEX \ \exists \ x \in \text{PEX } \textit{s.t. } op \in x$$

Intuitively, the previous theorem states that if an operator is picky according to *NedExplain*, then it (or more precisely its condition) can be found in some term in the polynomial answer of *Ted*. However given the set of picky operators in NEX, it is not guaranteed that

1. there exists one term in *PEX* that contains all operators in NEX, or
2. each operator in NEX is a term (by iteself) in *PEX*.

This is entailed by the fact that in *NedExplain*, as soon as the trace of a compatible tuple is lost at a certain node, the remaining nodes remain unchecked for this specific compatible tuple. Even worse, when at some node the trace of all compatible tuples disappears the procedure stops and the rest of the nodes remain unchecked.

So, to conclude we cannot ensure that *NedExplain*, given one query tree $\mathcal{T}$, computes all the picky subqueries corresponding to the conditions involved in the terms in *PEX*.

Example 4.2.23. *Consider again the query tree in Figure 4.20. According to* NedExplain *algorithm, we obtain NEX=$\{op_1, op_3\}$. From Example 4.2.21 we have* PEX=$op_3 + 2op_2op_3 + op_3op_4 + op_1 + 2op_1op_2 + op_1op_4$. *We can see that the first picky operator op_1 of NEX exists in three terms of* PEX*: op_1 and op_1op_2 and op_1op_4. The second picky operator is found in the rest of the terms. Thus, all the information that we obtain from NedExplain, we also obtain by* Ted.

Query minimization

Lemma 4.2.1 stated that isomorphic queries are equivalent w.r.t. *PEX*. Now, we show that this cannot be extended to equivalent queries in general by focusing on tableau minimization. Then, we investigate the relationship between the *PEX* of a query Q and the *PEX* of a minimized query Q' w.r.t. Q. For this discussion, and to be able to perform tableau minimization, we restrict our attention to the class of conjunctive queries without inequalities under set semantics.

The equivalence of two queries Q and Q' relies on the homomorphism theorem [CM77] in order to prove the containment in both directions. Recall that, for tableau queries Q and Q', $Q' \subseteq Q$ iff there exists a homomophism $h : Q \to Q'$. A query tableau Q can be minimized to an equivalent Q', by deleting redundant rows.

Definition 4.2.11. *(Minimized tableau) Let Q and Q' be equivalent query tableaux. Then, Q' is a minimized tableau w.r.t. Q if the set of rows of Q contains the set of rows of Q'.*

When Q' is a minimized query w.r.t. Q, the homomorphism h maps the variables of the tableau Q to variables of the tableau Q'. Intuitively, since rows are eliminated during minimization, and each row of Q has unique variables by Definition 2.1.2, the variables in the eliminated rows are also eliminated. Thus, the variables of Q can be split into two sets: (i) X containing the variables that are mapped to themselves through h and, (ii) X_{elim} containing the variables that are mapped to other variables than themselves and thus do not appear in the minimized tableau Q'.

If for a query we cannot find an equivalent one with less rows, then it is *minimal.* Our discussion focuses on any minimized query and not only on the minimal one.

Example 4.2.24. *Consider again the instance and query Q in Figure 4.19 (a) and (b). For convenience, we repeat the query Q along with its equivalent query Q' in Figure 4.22.*

The tableau Q' is a minimized tableau w.r.t. Q because the two queries are equivalent and Q' has one less row than Q. To prove the containment $Q' \subseteq Q$ we can exhibit a homomorphism $h : Q \to Q'$, s.t.

$$\begin{gathered}
h(a_1) = a_1 \\
h(x_1) = x_1 \\
h(x_2) = x_2 \\
h(x_3) = x_3 \\
h(x_4) = x_4 \\
h(x_5) = x_5 \\
h(x_6) = x_1 \\
h(x_7) = x_2 \\
h(x_8) = x_3 \\
h(s_Q) = s_{Q'}
\end{gathered}$$

Thus,

$$X{=}\{x_1, x_2, x_3, x_4, x_5\}$$

	$R_1.A$	$R_1.B$	$R_1.C$	S.A	S.B	$R_2.A$	$R_2.B$	$R_2.C$	C_Q
R_1	x_1	x_2	x_3						$x_1 = a_1 \wedge x_2 = x_5$
S				x_4	x_5				$x_2 = x_5 \wedge x_7 = x_5$
R_2						x_6	x_7	x_8	$x_6 = a_1 \wedge x_7 = x_5$
s_Q			x_3	x_4					

(a) Query Q

	$R_1.A$	$R_1.B$	$R_1.C$	S.A	S.B	$C_{Q'}$
R_1	x_1	x_2	x_3			$x_1 = a_1 \wedge x_2 = x_5$
S				x_4	x_5	$x_2 = x_5$
$s_{Q'}$			x_3	x_4		

(b) Minimized query Q'

Figure 4.22: Example query Q and minimized query Q'

and

$$X_{elim}=\{x_6, x_7, x_8\}$$

For the containment $Q \subseteq q'$, the homomorphism is the identity function for all variables and constants in Q'. Moreover, the conditions of the two tableaux (as named in Figure 4.19 (d)) are mapped as follows: $h(op_1^Q) = op_1^{Q'}$, $h(op_2^Q) = op_1^{Q'}$, $h(op_3^Q) = op_3^{Q'}$ and $h(op_4^Q) = op_3^{Q'}$.

In this case Q' cannot be further minimized, thus Q' is minimal.

Theorem 4.2.3 states that Q subsumes Q' w.r.t. *PEX*.

Theorem 4.2.3. *Let $\mathcal{S}_Q$ be a database schema and $\mathcal{I}_Q$ be an instance over $\mathcal{S}_Q$. Let Q be a query over $\mathcal{S}_Q$ and Q' a minimized query w.r.t. Q. Let vWN be a Why-Not question and assume* PEX_Q=PEX(Q, *vWN*, $\mathcal{I}$) *and* $\text{PEX}_{Q'}$=PEX(Q', *vWN*, $\mathcal{I}$). *Then,*

1. $\forall y \in \text{PEX}_{Q'}$, $\exists x \in \text{PEX}_Q$ *s.t.* $y \in x$
2. $\forall x \in \text{PEX}_Q$, $\exists y \in \text{PEX}_{Q'}$ *s.t.* $y \in x$

The first point in Theorem 4.2.3 states that every term in the Why-Not answer polynomial of Q' is contained in a term of the Why-Not answer polynomial of Q. This result is expected since all conditions in Q' are also conditions of Q. The second point states the inverse: every term in the Why-Not answer polynomial of Q can be mapped to a term in the Why-Not answer polynomial of Q'. Intuitively, this means that the conditions of Q eliminated by minimization resulting in Q' cannot exist alone (i.e., without a non-eliminated condition) in one picky condition set. For example, since the join op_4=$x_5 = x_7$ in Q (see Figure 4.22) is eliminated through tableau minimization, then $=op_4$ is not a term of the Why-Not answer polynomial for Q.

In the following discussion, we sketch a proof of the previous theorem going through each constitutive part of *PEX*:

1. Why-Not question and compatible tuples, and
2. explanations.

	$R_1.A$	$R_1.B$	$R_1.C$	S.A	S.B	$R_2.A$	$R_2.B$	$R_2.C$	vWN
R_1	x_1	x_2	x_3						$x_3 = c_2$
S				x_4	x_5				$x_4 = a_2$
R_2						x_6	x_7	x_8	
s	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	

T_{vWN}

	$R_1.A$	$R_1.B$	$R_1.C$	S.A	S.B	vWN
R_1	x_1	x_2	x_3			$x_3 = c_2$
S				x_4	x_5	$x_4 = a_2$
s'	x_1	x_2	x_3	x_4	x_5	

T'_{vWN}

Figure 4.23: Compatibility tableaux for Q and Q'

We consider that Q' is a minimized query w.r.t. Q and that Q'=$h(Q)$, where h is a homomorphism. Moreover, it is indirectly shown that equivalent queries are not equivalent w.r.t. *PEX*, as the running example shows.

Why-Not question and compatible tuples Queries Q and Q' are equivalent, thus have the same summary rows depicting their output schemas. Let vWN be a Why-Not question for query Q. Since vWN is defined over the output schema of Q, which is the same as the output schema of Q', it follows that vWN is a well defined Why-Not question for the query Q' as well.

The tableau T_{vWN} is defined based on the Why-Not question vWN and the body of the query Q, and thus it is expected to be different for different queries. Nevertheless, T'_{vWN} for Q' can be built using T_{vWN} as stated by the following Lemma.

Lemma 4.2.3. *(From T_{vWN} to T'_{vWN})*
If T_{vWN}=$(var(T_Q), T_Q, vWN)$ is the compatibility tableau for Q and vWN then T'_{vWN}=$(h(var(T_Q)), h(T_Q), vWN)$ is the compatibility tableau for Q' and vWN.

For our example, T_{vWN} and T'_{vWN} are shown in Figure 4.23. By definition, the summary of the tableau T_{vWN} contains all variables of Q. Recall that the variables of Q are split into two sets X and X_{elim}, thus s=$var(T_Q)$=XX_{elim}.

To compute the compatible tuples w.r.t. vWN for Q and Q', we need to evaluate T_{vWN} and T'_{vWN} respectively over $\mathcal{I}$. Let f be a valuation of T_{vWN} over the instance $\mathcal{I}$. For the summary s' of the tableau T'_{vWN} it holds that:

$$f(s') = f(h(X)) = f(X)$$

Also,

$$f(s) = f(XX_{elim}) = f(X)f(X_{elim}) = f(s')f(X_{elim})$$

Taking into consideration this last equation and that the Why-Not question is not defined over X_{elim} variables, it can be deduced that the set of compatible tuples CT corresponding to the query Q can be mapped to the set of compatible tuples in CT' using the onto relation

$$\zeta : CT \to CT' \ s.t. \ \forall \tau \in CT : \zeta(\tau) = \tau' \ s.t. \ \tau' {=} \tau_{|X}$$

where $\tau_{|X}$ is the restriction of τ to attributes in X.

CT	$\mathcal{E}_\tau$
$\tau_1 = a_1 b_2 op2 a_2 b_1 \quad a_3 b_1 op1$	$\{op_2, op_3\}$
$\tau_2 = a_1 b_2 op2 a_2 b_1 \quad a_1 b_2 op2$	$\{op_3, op_4\}$
$\tau_3 = a_1 b_2 op2 a_2 b_1 \quad a_1 b_1 op3$	$\{op_3\}$
$\tau_4 = a_1 b_2 op2 a_2 b_1 \quad a_2 b_1 op2$	$\{op_2, op_3\}$
$\tau_5 = a_2 b_1 op2 a_2 b_1 \quad a_3 b_1 op1$	$\{op_1, op_2\}$
$\tau_6 = a_2 b_1 op2 a_2 b_1 \quad a_1 b_2 op2$	$\{op_1, op_4\}$
$\tau_7 = a_2 b_1 op2 a_2 b_1 \quad a_1 b_1 op3$	$\{op_1\}$
$\tau_8 = a_2 b_1 op2 a_2 b_1 \quad a_2 b_1 op2$	$\{op_1, op_2\}$

CT'	$\mathcal{E}_{\tau'}$
$\tau'_1 = a_1 b_2 op2 a_2 b_1$	$\{op_3\}$
$\tau'_2 = a_2 b_1 op2 a_2 b_1$	$\{op_1\}$

Figure 4.24: Compatible tuple sets $CT(T_{vWN}, \mathcal{I})$ and $CT(T'_{vWN}, \mathcal{I})$ and picky conditions sets $\mathcal{E}$ and $\mathcal{E}'$

Moreover, it holds that $CT = CT' \times \mathcal{I}_{elim}$, where $\mathcal{I}_{elim}$ is obtained by cross product over the relations that correspond to rows in Q that are not in Q' (and thus contain the variables X_{elim}).

This means that there are $|\mathcal{I}_{elim}|$ more compatible tuples for Q than for Q', each one of which will have $|X_{elim}|$ times more attributes. Consequently, it is more efficient to deal with the minimized query, as not only has it less operators to check but also it generates less and more concise compatible tuples.

Figure 4.24 shows the sets of compatible tuples CT and CT' for our example (ignore the $\mathcal{E}_\tau$ column for the moment). It is obvious that $CT' = CT \times \mathcal{I}_{|R_2}$.

Explanations and Why-Not answer Next, we identify the explanations $\mathcal{E}_\tau$ for each compatible tuple and further compose the Why-Not answer. Finally, we establish the relationship between the Why-Not answer of Q and Q' as stated in Theorem 4.2.3.

For the first item of Theorem 4.2.3, it is ease to see that any 'minimized' compatible tuple τ' is part of some compatible tuple τ. As a consequence, the explanations responsible for pruning τ' are also responsible for pruning τ (along with possibly more conditions not satisfied by other parts of the tuple τ).

For the second item, we extend the onto mapping $\zeta : CT \to CT'$ to map the explanation $\mathcal{E}_\tau$ for $\tau \in CT$ to an explanation $\mathcal{E}_{\tau'}$ for $\tau' \in CT'$ as follows:
if $\zeta(\tau) = \tau'$ then $\zeta(\mathcal{E}_\tau) = \mathcal{E}_{\tau'}$ if $\mathcal{E}_{\tau'}$ contains only these conditions in $\mathcal{E}_\tau$ that are expressed over non-eliminated attributes.

Example 4.2.25. *Consider the compatible tuple* $\tau_1{=}(a_1, b_2, c_2, a_2, b_1, a_3, b_1, c_1) \in CT$ *and the compatible tuple* $\tau'_1{=}\zeta(\tau){=}(a_1, b_2, c_2, a_2, b_1) \in CT'$*, with the tableaux* T_{τ_1} *and* $T_{\tau'_1}$ *(Table 4.13) respectively. The explanations for* τ *and* τ' *are* $\mathcal{E}_{\tau_1} = \{op_3, op_2\}$ *and* $\mathcal{E}_{\tau'_1} = \{op_3\}$*. Note that since* τ'_1 *is part of* τ_1 *indeed* $\mathcal{E}_{\tau'_1} \subseteq \mathcal{E}_\tau$ *(point 1 of Theorem 4.2.3). Moreover,* $\zeta(\mathcal{E}_{\tau_1}) = \mathcal{E}_{\tau'}$ *(point 2 of Theorem 4.2.3).*

Figure 4.24 links every compatible tuple with its picky conditions. Summing the explanations for the compatible tuples in CT and CT', we obtain the following Why-

Not answers w.r.t. Q and Q' respectively:
$\mathrm{PEX}_Q = op_3 + 2op_2op_3 + op_3op_4 + op_1 + 2op_1op_2 + op_1op_4$ *and*
$\mathrm{PEX}_{Q'} = op_3 + op_1$.

Note that we can map each term in PEX_Q to one term in $PEX_{Q'}$, and each term in $PEX_{Q'}$ to a term in PEX_Q.

Table 4.13: Tableau $T_{\mathcal{T}_1}$ and $T'_{\mathcal{T}_1}$

	$R_1.A$	$R_1.B$	$R_1.C$	S.A	S.B	$R_2.A$	$R_2.B$	$R_2.C$	$cond_{\mathcal{T}}$	C_Q
R_1	x_1	x_2	x_3						$x_1 = a_1, x_2 = b_2, x_3 = c_2$	$x_1 = a_1, x_2 = x_5$
S				x_4	x_5				$x_4 = a_2, x_5 = b_1$	$x_2 = x_5, x_5 = x_7$
R_2						x_6	x_7	x_8	$x_6 = a_3, x_7 = b_1, x_8 = c_1$	$x_6 = a_1, x_5 = x_7$

	$R_1.A$	$R_1.B$	$R_1.C$	S.A	S.B	$cond_{\mathcal{T}'}$	$C_{Q'}$
R_1	x_1	x_2	x_3			$x_1 = a_1, x_2 = b_2, x_3 = c_2$	$x_1 = a_1, x_2 = x_5$
S				x_4	x_5	$x_4 = a_2, x_5 = b_1$	$x_2 = x_5$

***PEX* approximation** As said before, computing the Why-Not answer for a minimized query Q' is more efficient than computing the one for Q. One could argue that finding a minimized query for Q is a complex procedure, however the literature [CR97] shows that it can be considered practically, despite its theoretical complexity. Moreover, the discussion until now did not consider the minimal query but a minimized query for Q, which renders the problem practically more reasonable.

For this reason, we investigate now how we can obtain an approximate *PEX* for the query Q when *PEX* is available for a minimized Q' w.r.t. Q.

Definition 4.2.12 (Approximate *PEX*)**.** *Let Q be a query and Q' be a minimized query w.r.t. Q. Let us consider the tableau homomorphism h from Q to Q'. Assume that* $\mathrm{PEX}_{Q'}$=PEX($Q', vWN, \mathcal{I}$) *is known. Then, the approximate* PEX *of Q is*

$$\mathrm{PEX}_{Appr} = \sum_{\mathcal{E}' \textit{ in } \mathrm{PEX}_{Q'}} \left(\prod_{op \in h^{-1}(\mathcal{E}')} op \right)$$

where

$$h^{-1}(\mathcal{E}') = \{op \mid op \in h^{-1}(op') \wedge op' \in \mathcal{E}'\}$$

and

$$h^{-1}(op') = \{op \mid h(op) = op' \wedge op \in C_Q\}$$

In the next theorem, it will be clear that the approximate *PEX* concerns only the *terms* of the polynomial answer and that no estimation can be provided for the *coefficients.*

Theorem 4.2.4. *Let* PEX_Q *and* PEX_{Appr} *be the Why-Not answer and the approximate Why-Not answer w.r.t. Q respectively. Then,*

$$\forall \mathcal{E} \in \text{PEX}_Q \quad \exists \mathcal{E}_{Appr} \in \text{PEX}_{Appr} \ s.t.\ \mathcal{E}_{Appr} \subseteq \mathcal{E}$$

Example 4.2.26. *Let us assume now that the queries introduced in Figure 4.22 are changed so as the selection conditions be* $x_1 = a_5$ *and* $x_6 = a_5$*, while we assume the same Why-Not question vWN. Considering the minimized query* Q'*, its Why-Not answer is* $\text{PEX}_{Q'} = op_1op_3{+}op_1$*. As* $h^{-1}(op_1) = \{op_1, op_2\}$ *and* $h^{-1}(op_3) = \{op_3, op_4\}$ *the approximate Why-Not answer for Q is* $\text{PEX}_{Appr} = op_1op_2op_3op_4{+}op_1op_2$*. Indeed,* $\text{PEX}_Q = 3op_1op_2op_3 + op_1op_2op_3op_4 + 3op_1op_2 + op_1op_2op_4$*. It is evident that all the terms in* PEX_Q *are supersets of some term of* PEX_{Appr}*. However, no estimation of the coefficients is provided.*

4.2.8 Conclusion

This section provided the *Ted* framework for answering Why-Not questions with query based explanations formalized as polynomials with query conditions as variables and integer coefficients. Opposed to previous works on query-based explanations, the Why-Not answer polynomial captures all possible and complete explanations for the missing answers from the result of a query Q by disengaging from the query-tree approach. The Why-Not answer polynomial is an informative expression, representing not only which query conditions are responsible for the missing tuples but also how they are responsible, i.e., in which combinations (represented by the polynomial terms). Furthermore, the coefficients of the Why-Not answer polynomial suggest an upper bound for the number of recovered missing tuples following a fixing of some condition combination.

The *Ted* framework including the definition of Why-Not answer polynomial enables to consider Why-Not questions for relational databases under set, bag and probabilistic semantics in a unified way. Moreover, we have studied the Why-Not answer polynomial in the context of different equivalent query classes. More specifically we argue that isomorphic conjunctive queries with inequalities have isomorphic Why-Not answer polynomial w.r.t. isomorphic Why-Not questions. Then, we show that this is not the case for queries resulting from tableau minimization, for which case an approximation of the Why-Not answer is proposed.

Furthermore, we have introduced two algorithms to compute Why-Not answer polynomials: *Ted* and the more efficient *Ted++* algorithms. The latter algorithm's 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 answer for a developer. Also, we showed that *Ted++* scales well with various parameters, making it a practical solution.

The class of queries considered in the study is conjunctive queries with inequalities while both simple and complex Why-Not questions are considered. We showed that extending the framework to unions of conjunctive queries and general Why-Not questions is trivial. However, an extension for adding aggregation to the query operators set requires more investigation, as extending the current framework would require to consider all the possible subsets of the input database tuples rendering the method impractical. On the other side, extending the queries to include negation, is feasible if we consider the right part of the negation as a relation, i.e., a set of tuples. In this case, the negation condition could be applied as a condition. However, this trivial extention can be applied only to the naive *Ted* Algorithm. The *Ted++* algorithm needs more consideration an possibly redesigning to include negation.

The polynomial has the benefit of being an elegant formalism that can subsequently be used for further processing. Example applications would be ranking the importance of "misbehaving" query operators in the query, computing query refinements to recover missing tuples, or estimating the minimum number of side-effects of a refinement, etc. In the next chapter we show how we use the Why-Not answer polynomials to address the query refinement problem.

4.3 Summary

In this chapter we presented two frameworks computing query based explanations to Why-Not questions given a query and thus aiding the query debugging experience of the end user. The main characteristic of the first algorithm, *NedExplain*, is that it is based on a query tree representation, while *Ted* reasons on the the query statement. *NedExplain* can handle a wider class of queries and is more efficient than *Ted*. On the other side, *Ted*'s main strengths are that it can handle a wider class of Why-Not questions and that it provides a complete set of query based explanations w.r.t. different query tree re-orderings as opposed to *NedExplain*. Here complete is twofold, meaning that all explanations are identified and that each explanation is in itself complete. The efficiency problem entailed by *Ted* has been addressed in the optimized *Ted++* algorithm, which allows for computing a Why-Not answer polynomial in time comparable with the time needed for *NedExplain* to compute the respective query-based explanation.

Finally, we discussed that the Why-Not answer polynomial resulting from *Ted* (or *Ted++*) provides a unified setting for Why-Not provenance for set, bag and probabilistic data model semantics. We also provided a way for computing an approximate Why-Not answer polynomial w.r.t. a query, when a minimized version of the query is available.

Overall, we argue that the Why-Not answer polynomial approach for modelling query-based explanations and the *Ted++* algorithm for producing Why-Not answer polynomial provide a more complete and 'secure' framework for query debugging, as the user can be sure that she gets all the ways in which he loses expected tuples. This is important when she needs to choose the conditions to repair in a subsequent query fixing task, as we will see in the next chapter. However, even if *NedExplain* provides an incomplete set of query-based explanations, it still could be useful in an interactive setting. In such a setting, the user progressively repairs a query, by fixing one operator at a time, driven by the outcome of previous operator fixes.

Chapter 5

Query Refinement Phase

In this chapter we describe our approach for refining a query in order to recover missing tuples in the query result. More specifically, in the previous chapter we have seen how to debug a query when we miss some tuples from its result. We have proposed two methods to produce query-based explanations, the one of which provided the explanations in the form of a polynomial. Now, we discuss how the Why-Not answer polynomial can be used in order to refine the query in such a way that it recovers missing tuples. To this end, we propose the *FixTed* algorithm that leverages the different explanations in the Why-Not answer polynomial in order to efficiently and effectively alter the query conditions that are indicated in each explanation. In this way, we compute various refinements, changing different combinations of erroneous query conditions. Usually users are most interested in refinements that do not differ a lot from their initial queries. Driven by this desideratum, we guarantee that among the computed refinements we return the one that has the highest similarity to the original query. Besides similarity on the query statement level, users care about how precise are the results of their queries, and prefer refinements that do not add many undesirable tuples in the original result set. As the refinements may be numerous, only the 'best' refinements are returned ranked based on a cost function that considers both similarity and precision, further described in the next sections.

The chapter is organised as follows. Section 5.1 gives the motivation for the problem we are addressing and describes in a high level the proposed solution. Section 5.2 outlines the main features of our approach and our contribution. Section 5.3 introduces the specialized problem for this chapter and relevant background notions. Section 5.4 continues with describing query refinements produced taking into account the explanations. Section 5.5 describes the steps of the *FixTed* algorithm towards producing refinement-based explanations. Section 5.6 shows that the framework provided by *Ted* and *FixTed* can be practically implemented in a platform to debug and fix queries in order to recover missing results. Finally, Section 5.7 summarizes and concludes the chapter.

Publications The query debugging and fixing platform [BHT15c] built on the algorithms *Ted++* (for debugging) and *FixTed* (for fixing) was published as a demonstration paper in the proceedings of the *Very Large Data Bases (VLDB) 2015* conference.

5.1 Motivation

After debugging a query, a developer has become aware of what conditions of the query are responsible for the missing tuples. So, if she wants to fix the query manually, she can focus her efforts on her preferred explanation. Take for example the airlines scenario in Example 4.2.1 on page 77. The *Ted* algorithm has provided the developer with the information that both the selection and the join of the query shown in Figure 4.10 form a query based explanation. Thus, she knows that if she tries to fix the query, she has to change both query conditions.

Knowing which query conditions to target helps developers to focus on relevant parts of the query during the fixing process. Still, the tedious task of actually rewriting the query remains. Intuitively, when a selection is too restrictive to let the expected tuples survive, the solution would be to relax this condition. Choosing to what extend to relax a condition is crucial: the desired tuples could still not survive the new condition if the relaxation was not sufficient, or on the other side too many 'irrelevant' tuples could be added into the result along with the desired ones, if we over-relax the condition. Then, when a join is the problem, was it because the developer actually should have used a left or right outer join instead of an inner join? And is such a change capable to produce the expected results?

Depending on the experience of the developer, her knowledge of the underlying dataset and of course her time availability, finding the optimal solution to the fixing problem may be a more or less demanding task. However, if we consider the number of possible changes that one can make to a single selection condition, it is evident that the problem becomes harder and harder to solve, as the number of conditions in a query-based explanation increases. Furthermore, computing a fix for each explanation adds more difficulty to the problem.

Clearly, fixing a query with respect to a Why-Not question is a tedious and time-consuming task. Therefore, we propose to semi-automatically support developers during query fixing by providing suggestions for possible query changes based on the $Why-Notanswerpolynomial$. More specifically, our techniques empower the developer to effortless obtain a query that generates the results that she wants, choosing among a variety of refinements the one(s) that best fits her preference, w.r.t. the syntax of the repaired query and its result. In the next sections, we describe how we can achieve this using our algorithm *FixTed* and the theoretical framework on which it is based.

5.2 Contribution

In this chapter, we make the following contributions.

Query refinements based on query-based explantions. We provide the framework to compute query refinements for conjunctive queries with inequalities. We are the first to leverage query-based explanations in the form of Why-Not answer polynomials to compute the query refinements. In this way we succeed to compute the most similar query refinements possible. Moreover, the polynomial provides us with the information which conditions to focus on, which allows us to reduce the search space for query refinements, ultimately leading to a more efficient solution (as opposed to an approach not considering query-based explanations).

Query refinements and explanation type A query-refinement obtained by our method is linked to an explanation (term) of the Why-Not answer polynomial. We distinguish between three main categories of query-refinements, depending on the type of the explanation. The first one considers explanations containing only selection conditions and in principal consists in conjunctive queries having the explanation selection conditions fixed. To obtain these, we proceed in two steps, exploiting the technique of skyline tuples, briefly outlined in Section 3.2.5 on page 37. More specifically, we propose a variation of skyline that potentially leads to tackling the selections refinement problem more efficiently. The first step is designed so as to perform the least changes to the selections in the explanations, leading to the most similar refinements to the query (per explanation). By changing the conditions proposed by each explanation we have as a consequence that not only missing tuples but also irrelevant tuples are added to the refined query result. To tackle this problem, we proceed to the second step that eliminates irrelevant (a.k.a. false positive) tuples from the refined result.

The second category of query refinements considers explanations with only join conditions. In this case, we introduce left or right outer joins in the query refinement and thus move to a wider class than conjunctive queries. We are not aware of other proposals generating such kind of refinements. The third category considers explanations with both join and selection conditions, and is a trivial extension of the previous categories.

FixTed Algorithm and EFQ Platform We provide an algorithm, *FixTed* to compute query refinements w.r.t a query and a Why-Not answer polynomial, as defined by our framework. *FixTed* also defines a ranking function to order the best refinements, which are in the skyline of queries w.r.t. similarity and precision. Then, we demonstrate the practicality of our proposal by incorporating the *Ted++* algorithm along with the *FixTed* algorithm into a novel platform named *EFQ*. *EFQ* provides the means for semi-automatically debugging and refining SQL queries, through an interactive interface.

5.3 Problem and Preliminaries

The problem we wish to tackle in this chapter is described in Problem Statement 2.3.1. As a reminder, given an explanation scenario $\Delta{=}(\mathcal{S}, \mathcal{I}, Q, WN)$, our goal is to find queries Q' recovering in their result $Q'[\mathcal{I}]$ missing tuples, while preserving all tuples from $Q[\mathcal{I}]$. Note, that we consider only conjunctive queries with inequalities in the input of the problem.

As already said, in our approach we assume available the query conditions that are responsible for the missing tuples. This knowledge is captured in the query-based explanation computed in the previous section. In order to be able to use the query-based explanations to fix the query, it is important to know not only which conditions are to blame, but also in which combinations they should be fixed so as to be able to recover missing tuples. In Section 4.2 we showed that the Why-Not answer polynomial (see Definition 4.2.2) captures all the possible condition combinations that prune out data relevant to the missing tuples. Thus, we use the Why-Not answer polynomial *PEX* as a starting point for our refinement process.

Taking into account this requirement, we restate Problem Statement 2.3.1 as follows.

Problem Statement 5.3.1. *Given a scenario* $\Delta{=}(\mathcal{S}, \mathcal{I}, Q, WN)$*, and the Why-Not answer polynomial* PEX *w.r.t.* Δ*, how can we efficiently compute a set of useful query refinements?*

Intuitively, we have already described what is a useful refinement for a user. On the one hand the original query statement and the refined one should not differ a lot. On the other hand, the refined query should not output many irrelevant tuples, that could be undesirable and not expected by the user.

We further specialize Definition 2.3.2 of query refinement to reflect the fact that they are linked to some query-based explanation of the polynomial *PEX*. As a result, each computed refinement outputs at least one missing tuple built from compatible tuples pruned out by the conditions of a specific explanation. As a reminder, the explanations are modelled as terms in a Why-Not answer polynomial. For example, in the Why-Not answer polynomial $2op_1op_2 + op_3$, there exist two explanations, the op_1op_2 and the op_3.

Definition 5.3.1 (Query refinement w.r.t. query-based explanation)**.** *Let* $Q{=}(\mathcal{S}, \Gamma, C)$ *be a conjunctive query,* CT *be the set of compatible tuples. Let* $\mathcal{E}$ *be an explanation from the Why-Not answer polynomial* PEX*. Then, a query refinement is a query* Q' *s.t.* $\exists t \in CT : \; t \models \mathcal{E} \wedge \pi_\Gamma[t] \in Q'[\mathcal{I}]$.

As mentioned in the contributions section, we develop different strategies for producing refinements depending on the type of the used explanation. In case of explanations that are *selections-only*, the refinements are conjunctive queries.

For refinements computed using *joins-only* explanations, we introduce left and right outer [RRS11], to replace the inner joins of the explanations (when possible as

Table 5.1: Tuples in the database instance $\mathcal{I}$, satisfying the condition $R.C{=}S.C$. Tuples marked with t are compatible tuples, with r are query result tuples and with u are irrelevant tuples.

$\sigma_{R.C=S.C}[\mathcal{I}]$

	A	B	C	D	E
t_1	1.6	5.6	2.8	1	5
t_2	7.5	2.4	8.4	7	5
t_3	6.4	4.9	3.6	7.2	5
t_4	1	9	1.8	3	5
t_5	2	6.8	6.7	3.7	5
t_6	2	5.6	5.2	8.3	5
t_7	5.8	8.5	8.5	5.2	5
t_8	9.2	1.8	1.6	8	5
t_9	11.5	3.5	5.8	1.4	5
t_{10}	9.5	4.2	7.6	6	5
t_{11}	10.1	6.4	7.6	1	5
r_1	1	2.4	7.5	5.2	1
r_2	2.1	2	2.8	6.1	2
r_3	1.6	1.1	6	6.8	3
u_1	1	6.2	3.8	2.9	6
u_2	3	4.2	0.8	9.2	8
u_3	5.7	7.5	8.2	3.1	8
u_4	7.1	4.9	1.2	5.8	7
u_5	6.4	2.8	3.4	4.4	7
u_6	9	3.2	9.2	1.4	4

discussed in Section 5.4.3). A left or right outer join is associated with a condition specified by $A\ \zeta_\theta\ A'$, where $\zeta{\in}\{⟕, ⟖\}$, $\theta \in \{=, <, \leq, >, \geq, \neq\}$ and A and A' are attributes. Note that the operators $⟕_{\theta'}$ (left outer join) and $⟖_{\theta'}$ (right outer join) are not commutative, thus

$$A\ \zeta_\theta\ A' \neq A'\ \zeta_\theta\ A$$

Note here that we use only left or right outer joins. Using a full outer join, where a right or left would suffice to output a missing tuple, would only yield less precise results. More details are given in Section 5.4.3

Finally, the mixed type of explanation yields refinements in the class of conjunctive queries enriched with the left and outer join operators.

Example 5.3.1 introduces the scenario for the running example of this chapter. This scenario will be refined step by step in the dedicated sections. At this point, it helps for describing the next background notion, i.e., the *skyline*, used in our refining method(s).

Example 5.3.1. *Consider the scenario* $\Delta{=}(\mathcal{S}, \mathcal{I}, Q, WN)$*, where*

1. $\mathcal{S}$=$\{R(A,B,C), S(C,D,E)\}$ *(for simplicity we omit here the attributes Id)*
2. Q=$(\mathcal{S}, \Gamma, \{A<5.5, B<4, R.C=S.C\})$, *where* Γ=$\mathcal{A}(\mathcal{S})$
3. WN=$\{S.E = 5\}$

For simplicity, in this example, we do not explicit the instance $\mathcal{I}$. *Moreover, we name the conditions:*

$$op_1 : A<5.5,\ \ op_2 : B<4,\ \ op_3 : R.C=S.C$$

So, the query condition set is $C_Q = \{op_1, op_2, op_3\}$.

The Why-Not question defines two partitions for the schema $\mathcal{S}_Q$, $Part_1$=$\{R\}$ *and* $Part_2$=$\{S\}$. *Since WN is defined only over S, S is a direct relation while R is an indirect relation.*

Then, assume that the query result consists of three tuples, prefixed with r

$$Q[\mathcal{I}]=\{r_1, r_2, r_3\}$$

and that the set of compatible tuples w.r.t. WN has thirteen tuples, prefixed with t

$$CT = \{t_1, \ldots, t_{12}\}$$

Finally, consider that the Why-Not answer polynomial has already been computed and is

$$\text{PEX} = 3op_1 + 4op_2 + 4op_1op_2 + 1op_3$$

We group the different explanations into two groups, based on the type of involved conditions:
W_1=$\{op_1, op_2, op_1op_2\}$ *(selections-only)*
W_2=$\{op_3\}$ *(joins-only)*

As we see from the polynomial, there are eleven compatible tuples $t_1, \ldots, t_{11}$ *pruned out by selections-only explanations (i.e.,* W_1*) and one tuple* t_{12} *pruned out by joins-only (i.e.,* W_2*).*

In our approach, as will be made clear in the next section, we are only interested in the compatible tuples pruned out by explanations involving selection conditions. In our example, these are the eleven tuples $t_1, \ldots, t_{11}$. *We observe that these tuples must satisfy the join condition* $R.C = S.C$ *(i.e.,* op_3*) of the query, because they were not pruned out by* op_3. *Consequently, they are part of the result set of the query consisting only of the condition* op_3 *and whose result is presented in Table 5.1.*

Table 5.1 displays tuples as follows: firstly it displays the compatible tuples $t_1, \ldots, t_{11}$, *then the result tuples* $r_1, \ldots r_3$ *and finally the irrelevant tuples* $u_1, \ldots, u_6$, *meaning neither result tuples nor compatible tuples. Figure 5.1 displays graphically the tuples of Table 5.1 as points in the space defined by the attributes* A, B, C, D, E.

Assume also that the compatible tuple pruned out by op_3 *is* $t_{12}=(R.A:2, R.B:2, R.C:3, S.C:4, S.D:6, S.E:5)$.

The questions now is how can we fix the query Q *in order to obtain some tuples with* $E:5$ *in the result set. The answer is provided gradually in the flow of the discussion in this chapter.*

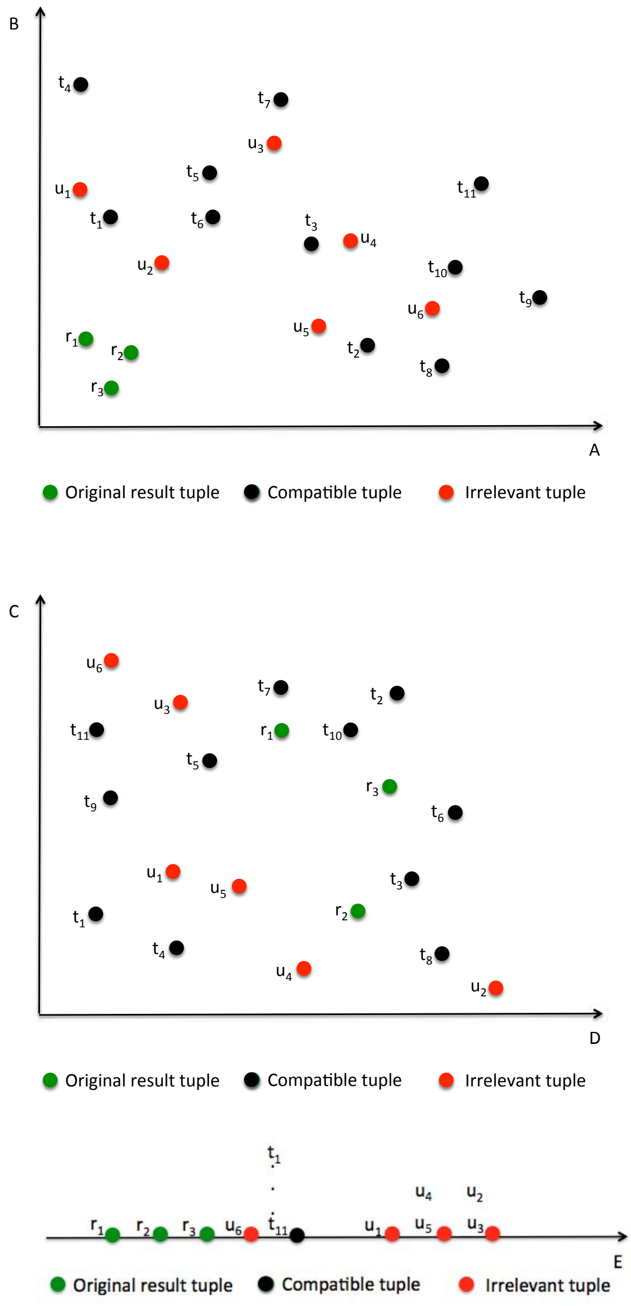


Figure 5.1: Tuples from Table 5.1 displayed in the dimension space A, B, C, D, E.

Before continuing with the description of the different kinds of query refinements and how we produce them, we conclude this introductory section by discussing the central notion of *skyline* [BKS01, ZDTT10, ST12, PTFS03, GM15] (also known as the *maximum vector problem* [KLP75]) in our context.

Skyline compatible tuples

When a selection condition is too restrictive to let a compatible tuple pass (and generate a missing tuple), then intuitively what we need to do is to relax this condition. When an explanation consists of more than one selection condition, then all these conditions need to be relaxed in the refined query.

To ensure that the refined query will yield the desired result, we have to choose the appropriate values for the new conditions. For this reason, all the new conditions should be permissive for at least one compatible tuple. The question now is which compatible tuple(s) we prefer to enact. We address this problem taking into consideration that the refined query should be as similar as possible to the original one. To accordingly prune the search space of compatible tuples to be considered, we resort to the technique of *skyline*.

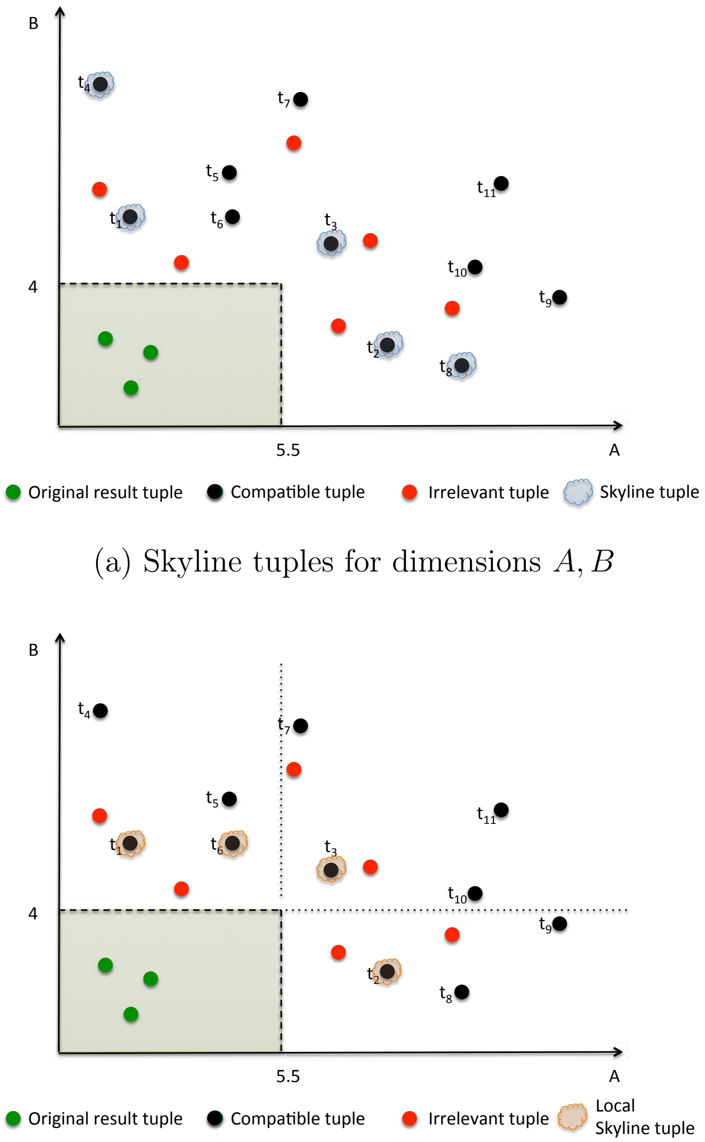
Intuitively, given a group of elements and a set of element properties (called *dimensions*), the skyline outlines the most interesting elements in this set, i.e., the elements that are for sure not worse than others w.r.t. the selected dimensions and a given preference. Traditionally, the elements are considered as points on the Cartesian space, with coordinates specified by the dimensions. To compare points we use the notion of *dominance* among the points.

Definition 5.3.2. *(k Dimensional Point Dominance) Let t_1 and t_2 be two points defined in k dimensions in set D. Then, we say that t_1 dominates t_2 (denoted $t_1 \succ_D t_2$) if*

1. *t_1 is better than or equal to t_2 (denoted $t_1 \succcurlyeq_D t_2$) in all dimensions, and*
2. *t_1 is better than t_2 (denoted $t_1 \succ_d t_2$) in at least one dimension d.*

In our case, we represent the compatible tuples as points whose dimensions are their attributes. Then, based on the values of their attributes we can decide which tuples are dominated or not by other tuples, provided some predefined preference.

Example 5.3.2. *Consider the tuples t_1 and t_6 from Table 5.1 and consider the dimensions specified by the attributes A and B. Assume a preference over lower values. Then, we say that t_1 dominates t_6 and write $t_1 \succ_{A,B} t_2$, because t_1 is better than t_6 in the dimension A ($1.6 < 2$) and it is equal to t_6 in the dimension B ($5.6 = 5.6$).*



(a) Skyline tuples for dimensions A, B

(b) Local Skyline tuples for A (fourth quarter), B (second quarter), and $\{A, B\}$ (first quarter)

Figure 5.2: Tuples in the Cartesian space and quarters defined by the query selection conditions.

The definition of skyline points in the specified dimensions is straightforward.

Definition 5.3.3. *(Skyline Point) Let $\mathcal{T}$ be a set of k dimensional points, with the dimensions specified by the set D. Then, $t \in \mathcal{T}$ is a* skyline point *if $\forall t' \in \mathcal{T}$ it holds that $t' \not\succ_D t$. The set of skyline points is called the Skyline of the set $\mathcal{T}$.*

Figure 5.2(a) is used to describe the skyline tuples (points), however for the moment we refer to it to describe point dominance and skyline. In this figure we project the tuples from Table 5.1 in the 2-dimensional space A, B. With green we mark the space restricted by the query selection conditions, and where the result tuples (marked with green) reside. Compatible tuples are marked with black and irrelevant tuples are marked with red. Please, ignore Figure 5.2(b) for the moment.

Example 5.3.3. *Consider the set of compatible tuples (points)* $\{t_1, \ldots, t_{11}\}$ *from Table 5.1 and consider the dimensions specified by the attributes* A *and* B*. If we consider that better means lower value, the skyline tuples (points) of this set are the points representing the tuples* t_1, t_2, t_3, t_4, t_8*. It can be easily verified that none of the tuples in* $\{t_1, \ldots, t_{11}\}$ *dominates any tuple in the Skyline, in these two dimensions. The skyline compatible tuples are marked as indicated in the caption in Figure 5.2(a).*

In our case, the preference is specified by the fact that we want values that are as close as possible to the ones in the original conditions of the explanation. So, given a certain explanation, we say that we prefer to use the compatible tuples that *for any* attribute involved in the conditions, result into a lower value of the following expression:

$$Diff_t = \mid v_t - v_{\mathcal{E}} \mid$$

where v_t is the value of the attribute specified by t and $v_{\mathcal{E}}$ the constant value to which the attribute is compared in the condition of the explanation.

Example 5.3.4. *Consider now the tuples* t_2 *and* t_3 *and the explanation* $\mathcal{E}$=$\{A<5.5\}$*. Then,* $v_{\mathcal{E}}$=5.5 *whereas for the tuples we have*

$$v_{t_2} = 7.5 \textit{ thus } Diff_{t_2} = \mid 7.5 - 5.5 \mid = 2$$

$$v_{t_3} = 6.4 \textit{ thus } Diff_{t_3} = \mid 6.4 - 5.5 \mid = 0.9$$

So, $t_3 \succ t_2$*, because* $Diff_{t_3} < Diff_{t_2}$.

We further define the signature of the function that computes the set of skyline tuples.

Notation 5.3.1. *(Skyline function) Let* $\mathcal{T}$ *be a set of tuples and* $\mathcal{A}tt$ *a set of attributes s.t.* $\mathcal{A}tt \subseteq \mathcal{A}(\mathcal{T})$*. Then,* $\mathcal{SL}(\mathcal{A}tt, \mathcal{T})$ *denotes the function that returns the set of skyline tuples in the set* $\mathcal{T}$ *under the dimensions defined by* $\mathcal{A}tt$*.*

Up to now we have been discussing how to use the set of compatible tuples in order to find the skyline tuples to be used in the query refinement process. However, it is not correct or beneficial to consider the whole set of compatible tuples, when considering one explanation. Indeed, using one explanation at a time enables us to restrict the quest of skyline tuples, locally in the space of compatible tuples pruned by the explanation only, as the following example demonstrates.

Example 5.3.5. *Let us consider again the set of compatible tuples* $\mathcal{T}$=$\{t_1, \ldots, t_{11}\}$*. Consider also the attributes* $\mathcal{A}tt$=$\{A, B\}$*. Then, the skyline tuples for these dimensions is*

$$\mathcal{SL}(\mathcal{A}tt, \mathcal{T}) = \{t_1, t_2, t_3, t_4, t_8\}$$

Figure 5.2(a) marks these tuples in the space defined by (A, B)*.*

The skyline technique presented here is used mainly for computing query refinements for selection-only explanations, as we will see in the next section. Additionally, we refer to this technique for 'screening' in a final step all the computed query refinements and reject those that are worse than others in a number of properties, defined later.

5.4 Query-Refinements

As we briefly mentioned before, for each explanation in the Why-Not answer polynomial we propose a number of refined queries and for each kind of explanation the class of the query refinement is different. When the explanation contains only selections, then the natural choice is to change (relax) the problematic conditions in a way that also missing tuples are permitted in the result. However, for joins such a solution would yield the introduction of a cross-product in the query, which is a naive solution that we consider too trivial. Moreover, cross-products rarely exist in applications, so such a refinement would not be interesting and useful. Therefore, for joins we propose the introduction of outer joins, when possible, in the place of the problematic joins. In this section we discuss in detail both cases and finally provide the algorithm to compute query refinements given one Why-Not answer polynomial.

5.4.1 Selections-Only explanations

For this type of explanations, associated with the set of explanations denoted by W_1, we split the process of computing query refinements in two phases. In the first phase, called *Minimum Distance Refinement (MDR)* we compute a set of query refinements per explanation. These refinements correspond to the minimum changes that we can make to the query based on one explanation. In the second phase, called *False Positive Elimination (FPE)* we further refine the set of queries obtained in MDR in order to eliminate tuples that are irrelevant to what the user defined as missing. We generally refer to such tuples as *false positive* tuples. Let us see now, the details for each query refinement set.

Minimum Distance Refinements

Let us consider that $\mathcal{E}$ is a selections-only explanation and that we wish to compute query refinements using $\mathcal{E}$. To relax the condition in $\mathcal{E}$ we rely on the values of the compatible tuples pruned by $\mathcal{E}$. However, since the compatible tuples may be numerous, the number of query refinements may be big. Moreover, one of the desiderata is that we change the constants involved in the conditions as little as possible, so that the changed condition mostly resembles the original one. In this sense, we can know a priori that some tuples will yield better refinements than others, because their values are ‘closer’ to the ones originally chosen by the user. This leads us to considering skyline tuples (Definition 5.3.1) in the set of compatible tuples pruned out by the considered explanation. The attributes to take into account also depend on the explanation; it is sufficient to consider only the attributes constrained by the conditions in the explanation, as we are only interested in the values of these attributes.

For this reason, we introduce at this point *local* skyline tuples w.r.t. an explanation $\mathcal{E}$. Besides considering attributes occurring in the explanations as explained before, we can further prune the space of considered tuples for the local skyline and thus the space of query refinements. The intuition is that given two query refinements $Q_{\mathcal{E}}$ w.r.t. $\mathcal{E}$ and $Q_{\mathcal{E}'}$ w.r.t. $\mathcal{E}'$ s.t. $\mathcal{E}'$ and $\mathcal{E}' \in W_1$ is a sub-explanation of $\mathcal{E}$ then if the changes on the common conditions are better in $Q_{\mathcal{E}'}$, then we prefer $Q_{\mathcal{E}'}$. So, we may discard the query refinement for $\mathcal{E}$. So, the local skyline tuples w.r.t. an explanation are defined as follows.

Definition 5.4.1. *Local skyline tuples w.r.t. an explanation*

Let $\mathcal{E}$ be an explanation, $CT_{\mathcal{E}}$ be the set of pruned compatible tuples by $\mathcal{E}$, and W_1 be the set of selections-only explanations. Consider the set of explanations

$$W_{sub} = \{\mathcal{E}' \mid \mathcal{E}' \in W_1 \wedge \mathcal{E}' \subseteq \mathcal{E}\}$$

Then, the set of local skyline tuples w.r.t $\mathcal{E}$ is the set

$$lSL_{\mathcal{E}} = \mathcal{SL}(\mathcal{A}(\mathcal{E}), CT_{\mathcal{E}} \setminus Dominated_{\mathcal{E}})$$

where

$$Dominated_{\mathcal{E}} = \{t \mid t \in CT_{\mathcal{E}} \text{ and } \exists \mathcal{E}' \in W_{sub} \text{ s.t. } t' \in lSL_{\mathcal{E}'} \text{ and } t' \succ_{\mathcal{A}(\mathcal{E}')} t\}$$

Example 5.4.1. *Let us revisit Example 5.3.5. Consider the selections-only explanation $\mathcal{E}_1$=$\{op_1\}$ and the compatible tuples CT_1=$\{t_2, t_8, t_9\}$ pruned out by op_1. The local skyline tuples w.r.t. $\mathcal{E}_1$ is*

$$lSL_{\mathcal{E}_1} = \mathcal{SL}(\{A\}, CT_1) = \{t_2\}$$

Figure 5.2(b) displays these tuples in the fourth (bottom-right) quarter of the diagram. Note that the quarters in this figure are defined by the conditions op_1 and op_2 of the query. The first (upper-right) quarter displays the tuples not satisfying neither op_1 nor op_2. The second (upper-left) quarter displays the tuples not satisfying op_2. The third quarter displays the query result, thus the tuples satisfying both op_1 and op_2. Finally, the fourth quarter displays the tuples not satisfying op_1 .

For $\mathcal{E}_2$=$\{op_2\}$ and $CT_2 = \{t_1, t_4, t_5, t_6\}$ we have that the local skyline tuples w.r.t. $\mathcal{E}_2$ is

$$lSL_{\mathcal{E}_2} = \mathcal{SL}(\{B\}, CT_2) = \{t_1, t_6\}$$

Figure 5.2(b) displays these tuples in the second quarter of the diagram.

Now, consider $\mathcal{E}_3 = \{op_1, op_2\}$ *and* $CT_3 = \{t_3, t_7, t_{10}, t_{11}\}$*. The set of sub-explanations of* $\mathcal{E}_3$ *is* $W_{sub} = \{\mathcal{E}_1, \mathcal{E}_2\} = \{\{op_1\}, \{op_2\}\}$*. We can see that the tuples* t_1*,* t_6 *from the local skyline of* op_2 *are dominating the tuple* $t_7 \in CT_3$*. Similarly, the tuple* t_2 *from the local skyline of* op_1 *dominates the tuples* t_{10} *and* t_{11}*. Thus, the only tuple left to be considered for the local skyline of* $\mathcal{E}_3$ *is* t_3*, and*

$$lSL_{\mathcal{E}_3} = \mathcal{SL}(\{A, B\}, CT_3 \setminus \{t_7, t_{10}, t_{11}\}) = \{t_3\}$$

Figure 5.2(b) displays these tuples in the first quarter of the diagram.

Each tuple in the set $lSL_{\mathcal{E}}$ of an explanation $\mathcal{E}$, yields one query refinement, obtained by using the values of the local skyline tuples to replace the values of the conditions in $\mathcal{E}$. If the comparison in the original condition is $<$ $(>)$, then it is changed to $\leq$ $(\geq)$. If it is $=$ it is changed to $\leq$ or $\geq$. Note that if the condition is $\neq$, then it is completely removed in the refined query. The query refinements obtained in this phase are defined as follows.

Definition 5.4.2. *(Minimum Distance Refined (MDR) Query) Let* W_1 *be the set of selections-only explanations and consider the sets of local skyline tuples* $lSL_{\mathcal{E}}$ *for each* $\mathcal{E} \in W_1$*. Then, the set of minimum distance refined queries* $\mathcal{Q}'_{mdr}$ *is*

$$\mathcal{Q}'_{mdr} = \{Q' \mid Q' \text{ refined query resulting from } t \in lSL_{\mathcal{E}}, \mathcal{E} \in W_1\}$$

Example 5.4.2. *The refined queries are*

with tuple t_2

$$Q'_{t_2} = (\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 7.5, S.B < 4, S.C = T.C\}$$

with tuple t_3

$$Q'_{t_3} = (\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 6.4, S.B \leq 4.9, S.C = T.C\}$$

The tuples t_1 *and* t_6*, the local skyline tuples for explanation* $B < 4$*, have the same value for the attribute* B*. So, they lead to the same refined query*

$$Q'_{t_1} = Q'_{t_6} = (\mathcal{S}_Q, \Gamma_Q, \{R.A < 5.5, S.B \leq 5.6, S.C = T.C\}$$

Thus, the set of minimum refined queries is $\mathcal{Q}'_{mdr} = \{Q'_{t_2}, Q'_{t_3}, Q'_{t_1}, Q'_{t_6}\}$*.*

We guarantee that $\mathcal{Q}'_{mdr}$ contains the refined query with the maximum similarity to the original query, when inequalities $(<, >, \neq)$ are involved. The proof is trivial, since we know from the polynomial what are the minimal possible condition changes and from the local skyline the most similar values to the original ones.

5.4.2 False Positive Elimination

The refinements in $\mathcal{Q}'_{mdr}$ are the result of a query relaxation process. Relaxing a query yields the addition of tuples that were not in the result of the original query. These extra tuples include (by design) some (projections of) compatible tuples, but also some irrelevant tuples.

To reduce the number of irrelevant tuples and so to obtain more precise refined queries, we move to the second refining phase concerning the queries in $\mathcal{Q}'_{mdr}$. In contrast with the query relaxation process, the new refined queries result from constraining the queries in $\mathcal{Q}'_{mdr}$, either by changing already existing conditions or by *adding* new conditions. The local skyline tuples again play a central role for the refining process, as the new conditions are generated based on their values.

Let $\mathcal{Q}'_{|\mathcal{E}} \subseteq \mathcal{Q}'_{mdr}$ be the set of refined queries associated with an explanation $\mathcal{E}$. Each query $Q' \in \mathcal{Q}'_{|\mathcal{E}}$ is associated with possibly more than one local skyline tuples in $lSL_{\mathcal{E}}$. We argue now that we can discard some of the local skyline tuples resulting in the same refined query Q', for the phase that eliminates false positive tuples. The following example illustrates this case.

Example 5.4.3. *In the previous example, we concluded to the same refined query for the tuples t_1 and t_6, because the tuples have the same value for the attribute B, defining the dimension of the skyline. Obviously, in the one-dimensional space defined by B, these tuples fall into the single point $B = 5.6$. However, if we consider the same tuples in the one-dimensional space defined by A, t_1 and t_6 are different points because $t_1.A < t_6.A$. Here, the objective is to produce as few false positive tuples as possible; thus the preference is determined by the fact that we want a smaller range of values to be permitted. Thus, (if we assume positive integers) t_1 is more 'dominant' than t_6 if we consider also the dimension A and as such, a refinement based on t_1 should be better than based on t_6.*

So, in order to avoid computing refinements that for sure contain more false positive tuples, we prune the space of local skyline tuples w.r.t. $\mathcal{E}$ keeping only *dominant* skyline tuples.

Definition 5.4.3. *Dominant local skyline tuple*

Let $\mathcal{E}$ be an explanation, and $lSL_{\mathcal{E}}$ its associated set of local skyline tuples. Let $\mathcal{J}$=$\{J_1, \ldots, J_n\}$, where n= $\mid \pi_{\mathcal{A}(\mathcal{E})}[lSL_{\mathcal{E}}] \mid$ be a partitioning of $lSL_{\mathcal{E}}$ s.t. a tuple t belongs in the partition J iff $\pi_{\mathcal{A}(\mathcal{E})}[t]$=$\pi_{\mathcal{A}(\mathcal{E})}(J)$.

Then, the set of dominant *local skyline tuples w.r.t. $\mathcal{E}$ is defined as*

$$dSL_{\mathcal{E}}= \bigcup_{i=1,\ldots,n} \{t \mid t \in J_i \text{ and } t \in \mathcal{SL}(\mathcal{A}(\mathcal{S}_Q) \setminus \mathcal{A}(\mathcal{E}), J_i)\}$$

Each dominant skyline tuple contributes in a different way to refining the queries in $\mathcal{Q}'_{mdr}$. Note, that the same query Q' may be changed in multiple ways, depending on how many dominant skyline tuples are associated with it.

Example 5.4.4. *In Example 5.4.1 we computed the sets of local skyline tuples for each explanation in W_1. For the explanations $\mathcal{E}_1$ and $\mathcal{E}_3$ the set of local skyline tuples contain one tuple so:*

$$dSL_{\mathcal{E}_1}{=}lSL_{\mathcal{E}_1}$$

$$dSL_{\mathcal{E}_3}{=}lSL_{\mathcal{E}_3}$$

For the explanation $\mathcal{E}_2$ though, the local skyline set contains t_1 and t_6, which have the same value for B. So, $\mathcal{J}{=}\{J_1\}$, where $J_1{=}\{t_1, t_6\}$. The set of attributes in $\mathcal{S}_Q$, not referred to in $\mathcal{E}$ is $Att{=}\{A, C, D, E\}$. From Table 5.1 and Definitions 5.3.2 and Notation 5.3.1 we can see that $t_1 \succ_{Att} t_6$. Thus, the set of dominant skyline tuples for explanation $\mathcal{E}_2$ is

$$dSL_{\mathcal{E}_2}{=}\{t_1\}$$

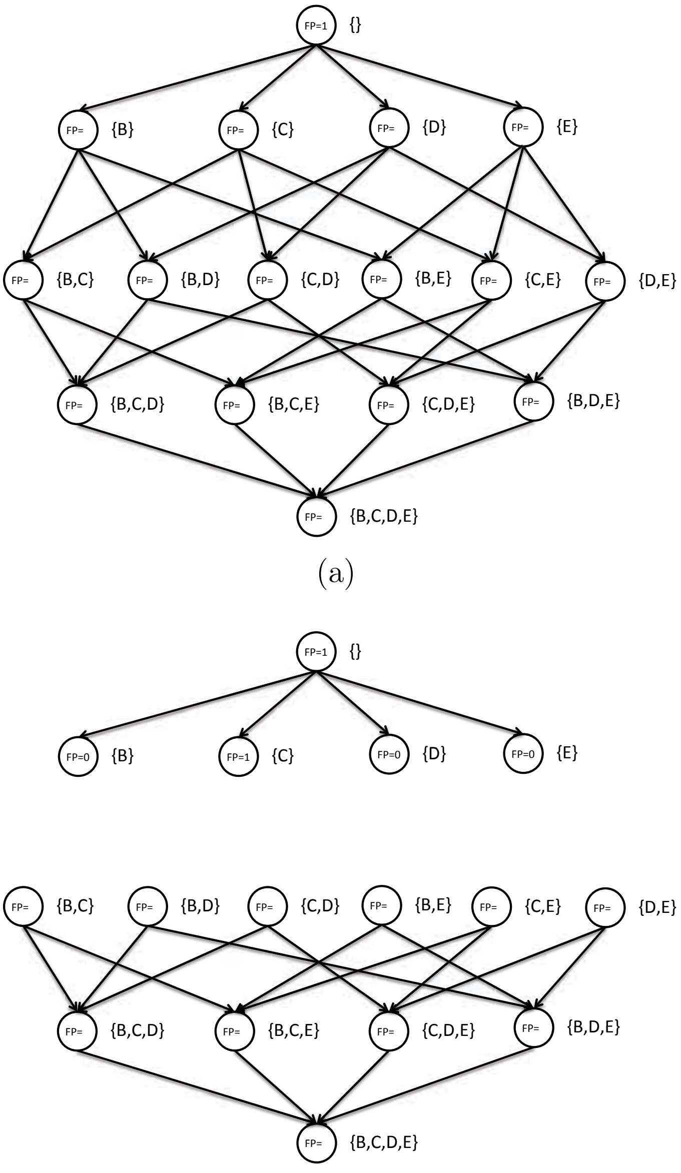
As one can easily guess, we may obtain a large number of different refined queries by using one dominant skyline tuple, depending on the number of involved attributes. The more conditions we add to the query, the more (or equal) irrelevant tuples we eliminate from the new query result but also the more we deteriorate the similarity w.r.t. the original query. For this reason, given a refined query Q' associated with an explanation $\mathcal{E}$ and a dominant skyline tuple t, we proceed to adding conditions to Q' progressively. In this way, we can control deteriorating the similarity of the query, when new conditions do not yield less irrelevant tuples.

The connection among the resulting (called *derivative*) query refinements of a query $Q' \in \mathcal{Q}_{mdr}$ using a dominant skyline tuple t and the explanation $\mathcal{E}$ is modelled as a rooted directed acyclic graph $G_{Q'}$ described as follows:

1. Each node N corresponds to a query refinement Q'' of Q' using t.
2. The root node r corresponds to the query Q'.
3. Each node has a unique label Att_N and a value FP s.t.
   — Att_N belongs to the powerset of $\mathcal{A}(\mathcal{S}_Q) \setminus \mathcal{A}(\mathcal{E})$.
   — FP is the number of false positive tuples of the associated Q''.
4. There exists a directed edge (N_1, N_2) from a node N_1 to a node N_2 if $Att_{N_2} \subset Att_{N_1}$ and $|Att_{N_2}| = |Att_{N_1}| - 1$.

Example 5.4.5. *Consider the refined query $Q'_{t_2} = \{A < 7.5, B < 4, R.C = S.C\}$ associated with the dominant skyline tuple t_2 and the explanation $op_1 : A < 5.5$. Then, the graph representing the derivatives of Q'_{t_2} is shown in Figure 5.3(a).*

When we have built the graph $G_{Q'}$ associated with t and $\mathcal{E}$, we ought to compute the query refinement Q'' associated with each node of the graph. Moreover, we need



(a)

(b)

Figure 5.3: Graph for derivatives of refined queries of Q'_{t_2} (a) initially, and (b) after edge pruning.

to find the number of false positive tuples that each Q'' outputs so as to later compute its precision. This number is called as the FP value of a node.

Assume we are at a node N. To proceed with the computations for N, we use the attributes Att_N of the label of N. Intuitively, we are going to create a new condition for each attribute in Att_N. We need to ensure that this condition is satisfied by all the query result tuples $Q[\mathcal{I}]$ and also the tuple t. The conjunction of the conditions will be used to form the refined query Q'' for node N. Then, we compute the number of false positive tuples output by the refined query Q'', by applying $C_{Q''}$ over the set of false positive tuples of the refined query Q' on the root node r

$$FP_N = \mid \sigma_{\bigwedge_{c \in C_{Q''}} c}[FP_r] \mid$$

where $\mathcal{FP}$ denotes the set of false positive tuples.

We distinguish among two categories of attributes in Att_N:

— Attributes also appearing in $\mathcal{A}(W_1)$.

— Other attributes (not appearing in $\mathcal{A}(W_1)$).

Depending on the category of the attribute, we proceed in a different way to create the condition on the attribute.

Attribute $A \in \mathcal{A}(W_1)$

If A is an attribute constrained by some selection condition op_A in the refined query Q, then we change the condition $op_A : A < a$ to

$$op'_A : A \leq max(\pi_A[Q[\mathcal{I}]], t.A)$$

In this way, we restrain the query in an attempt to eliminate false positive tuples by ensuring at the same time that we do not exclude any result tuples from $Q[\mathcal{I}]$.

Attribute $A \notin \mathcal{A}(W_1)$

If A is an attribute not originally appearing in the selection conditions of the query Q, we introduce the following condition on A

$$op'_A : min(\pi_A[Q[\mathcal{I}]], t.A) \leq A \leq max(\pi_A[Q[\mathcal{I}]], t.A)$$

By restricting the value of A between the minimum and maximum values, we wish to keep the briefest possible interval including all the interesting points and subsequently excluding as many false positive tuples as possible.

Example 5.4.6. *Consider the explanation Q'_{t_2} obtained in Example 5.4.2, which is associated with $\mathcal{E} = A < 5.5$ and the associated graph in Figure 5.3(a). There is one false positive tuple (u_5) in the result of Q'_{t_2}, thus the value of the root node is $FP = 1$.*

Then, let us investigate the node N with Att_N=$\{B, C\}$. Here, B falls in the first case and C falls in the second case.

For B, we create the condition

$$op'_B : B \leq max(\pi_B[Q[\mathcal{I}]], t_2.B)$$

which results in

$$op'_B : B \leq 2.4$$

For C, we create the condition

$$op'_C : min(\pi_C[Q[\mathcal{I}]], t_2.C) \leq C \leq max(\pi_C[Q[\mathcal{I}]], t_2.C)$$

which results in

$$op'_C : 2.8 \leq C \leq 8.4$$

Thus, we obtain the query

$$Q''{=}(\mathcal{S}_Q, \Gamma_Q, \{A \leq 7.5, B \leq 2.4, R.C = S.C, 2.8 \leq C \leq 8.4\})$$

to be associated with node N. The number of false positive tuples output by Q'' is zero; the tuple u_5 does not satisfy the condition op'_B.

In the same way we can compute the refined queries Q'' for each node in the graph of Figure 5.4(a).

To illustrate the exclusion of the false positive tuples from the refined query results in the cartesian space, consider for example the case of the refined query Q'_{t_3} associated with the explanation $op_1 op_2$. The result of this query is marked by a line in Figure 5.4(a). There are two false positive tuples in this result, u_2 and u_5. First consider the node of $G_{Q'_{t_3}}$ associated with $\{C\}$. Figure 5.4(b) illustrates the limits $op'_C : 2.8 \leq C \leq 7.5$ for the condition on C and shows that the false positive tuple u_2 is excluded from the result of the refined query Q'' associated with this node. Then, consider also the node associated with $\{D\}$. Figure 5.4(c) illustrates the limits $op'_D : 5.2 \leq D \leq 7.2$ for the condition on D and shows that both false positive tuples u_2 and u_5 are excluded from the result of the refined query Q'' associated with this node.

Remarks Even though every node in a graph $G_{Q'}$ represents a refined query Q'' of Q' that can be computed as previously described, it is possible to optimize the procedure in a two ways.

First of all, one may notice, that the nodes of the graph share attributes. Since the computed condition on an attribute is based on the query result tuples and the tuple t, this condition is always the same, regardless the node. So, we need to compute only once the condition for an attribute. Then, the same condition is used in every node it appears, to form the refined queries of Q'. For example, in the graph of Figure 5.4(a), the condition $op'_B : B \leq 2.4$ is used in the refined queries of all nodes with labels s.t. $B \subseteq Att_N$.

The second and most important remark concerns the edges of the graph, the use of which has not been made clear up to now. Indeed, we are going to prune edges outgoing from nodes whose FP value is null and we are going to only compute query refinements for those nodes connected though a path with the root. In this way, we avoid computing refinements adding conditions to other refinements already yielding zero false positive tuples.

For this reason, given a graph $G_{Q'}$ we visit the nodes in a breadth first search (BFS) manner. When for a node N the computed FP_N value is zero, then we disconnect

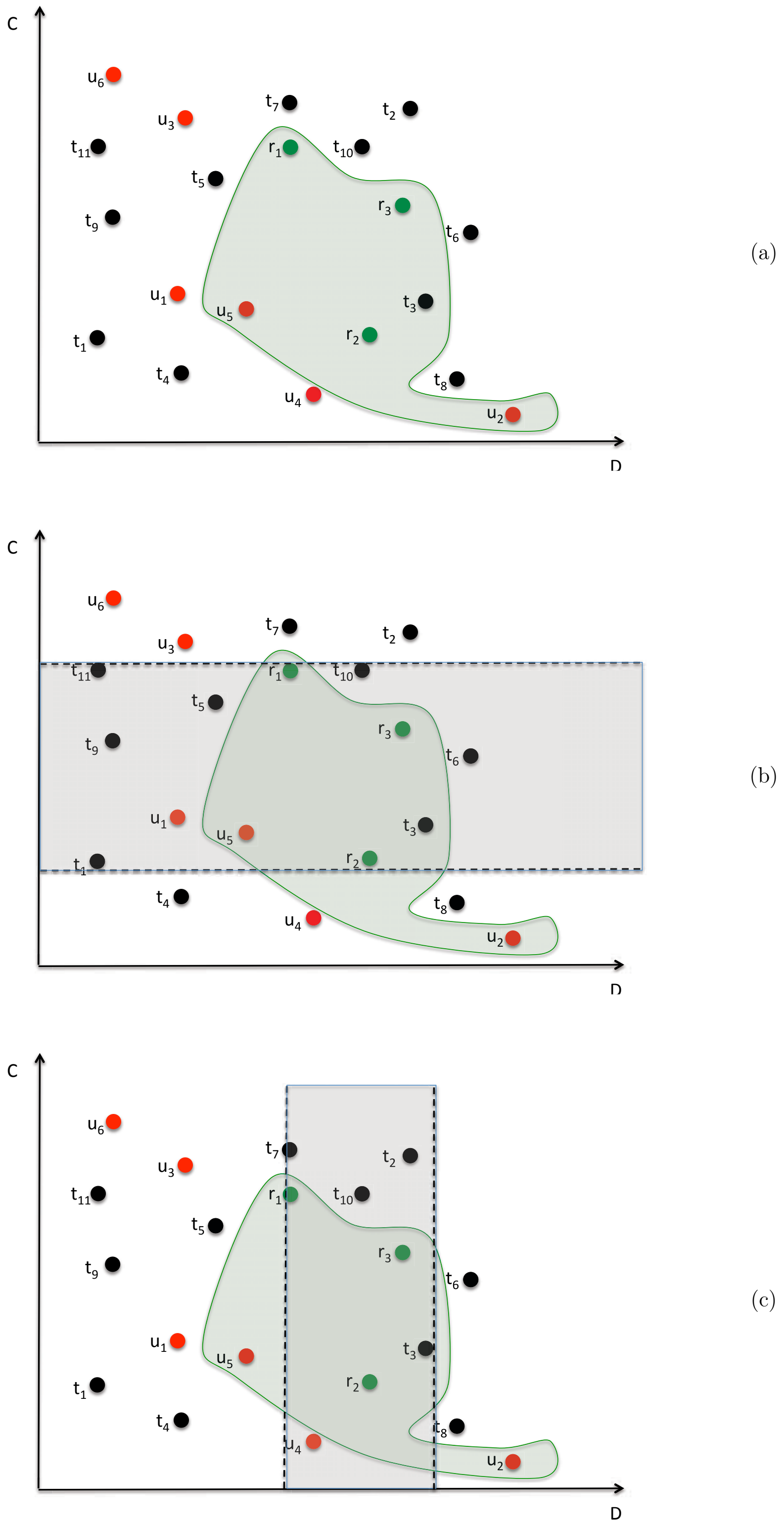


Figure 5.4: Representation in the *DC* space of the result tuples of refined query (a) Q'_{t_3}, (b) of refined query Q'' obtained by adding conditions on the attribute C, and (c) of refined query Q'' obtained by adding conditions on the attribute D.

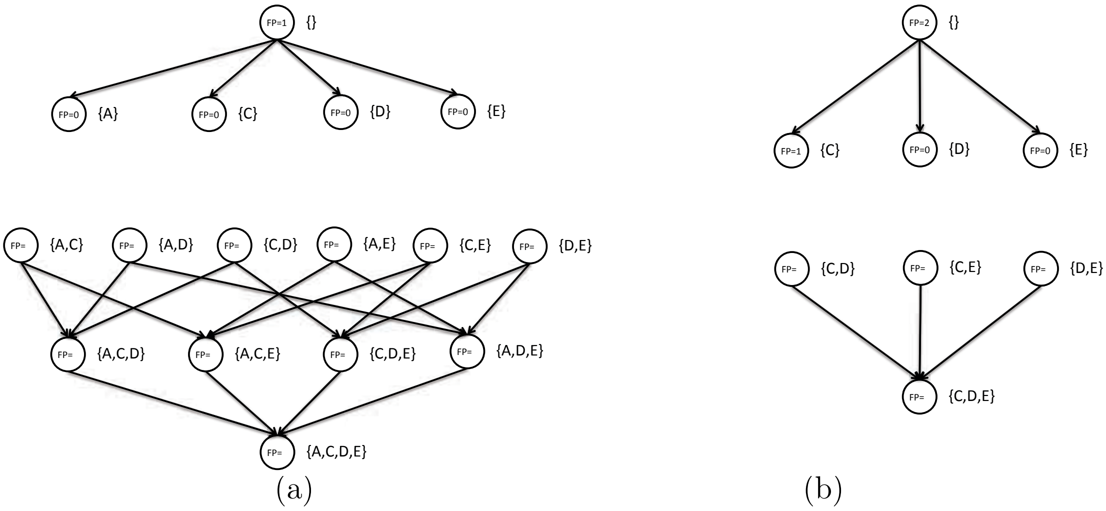


Figure 5.5: Graph for derivatives of refined queries of (a) Q'_{t_1}, and (b) Q'_{t_3}.

1. the edges of type (N, x), and
2. the edges of type (M, x) where x is a target node of some edge (N, x).

Example 5.4.7. *Consider again the graph in Figure 5.3(a). As we proceed in BFS order, we start with the node B. In the previous example, we saw that the false positive tuple u_5 does not satisfy the condition $op'_B : B \leq 2.4$ on attribute B. Thus, $FP_B = 0$, and we disconnect node B from its child nodes. We also disconnect these child nodes from all their parent nodes.*

We proceed with node C. The condition on C is $op'_C : 2.8 \leq C \leq 8.4$ which yields FP_C=1.

Then we visit node D, for which the condition is $op'_D : 5.2 \leq C \leq 7$. Thus, $FP_D = 0$, and we disconnect node D from its child nodes. We also disconnect these child nodes from all their parent nodes.

At this point, all nodes of the next level are disconnected. This means that there is no need to compute refinements for all the remaining nodes as the refining goal (to minimize false positive tuples) has already been achieved.

The resulting graph is shown in Figure 5.3(b).

In this same way we proceed with the graphs for Q'_{t_1} and Q'_{t_3}, and we obtain the graphs shown in Figure 5.5(a) and (b).

The resulting queries obtained in this phase are described as follows:

Definition 5.4.4. *(False Positive Elimination (FPE) Query Refinements) Let $\mathcal{Q}'_{mdr}$ be the set of MDR query refinements. Then, the set of false positive elimination query refinements $\mathcal{Q}'_{fpe}$ is*

$$\mathcal{Q}'_{fpe} = \bigcup_{Q' \in \mathcal{Q}'_{mdr}} \left(\bigcup_{N \in CC_{Q'}} Q''_N \right)$$

where $CC_{Q'}$ is the connected component containing the root node of the graph $G_{Q'}$ and Q''_N is the refined query associated with the node N.

Table 5.2: Refined queries for selections-only explanations for scenario of Example 5.3.1.

MDR	$Q'_{t_1}{=}(\mathcal{S}_Q, \Gamma_Q, \{R.A < 5.5, S.B \leq 5.6, S.C = T.C\}$
	$Q'_{t_2}{=}(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 7.5, S.B < 4, S.C = T.C\}$
	$Q'_{t_3}{=}(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 6.4, S.B \leq 4.9, S.C = T.C\}$
FPE	$Q''_{t_1,\{A\}}{=}(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 2.1, S.B \leq 5.6, S.C = T.C\}$
	$Q''_{t_1,\{C\}}{=}(\mathcal{S}_Q, \Gamma_Q, \{R.A < 5.5, S.B \leq 5.6, S.C = T.C, 2.8 \leq C \leq 7.5\}$
	$Q''_{t_1,\{D\}}{=}(\mathcal{S}_Q, \Gamma_Q, \{R.A < 5.5, S.B \leq 5.6, S.C = T.C, 1 \leq D \leq 6.8\}$
	$Q''_{t_1,\{E\}}{=}(\mathcal{S}_Q, \Gamma_Q, \{R.A < 5.5, S.B \leq 5.6, S.C = T.C, 1 \leq E \leq 5\}$
	$Q''_{t_2,\{B\}}{=}(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 7.5, S.B \leq 2.4, S.C = T.C\}$
	$Q''_{t_2,\{C\}}{=}(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 7.5, S.B < 4, S.C = T.C, 2.8 \leq C \leq 8.4\}$
	$Q''_{t_2,\{D\}}{=}(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 7.5, S.B < 4, S.C = T.C, 5.2 \leq D \leq 7\}$
	$Q''_{t_2,\{E\}}{=}(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 7.5, S.B < 4, S.C = T.C, 1 \leq E \leq 5\}$
	$Q''_{t_3,\{C\}}{=}(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 6.4, S.B \leq 4.9, S.C = T.C, 2.8 \leq C \leq 7.5\}$
	$Q''_{t_3,\{D\}}{=}(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 6.4, S.B \leq 4.9, S.C = T.C, 5.2 \leq D \leq 7.2\}$
	$Q''_{t_3,\{E\}}{=}(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 6.4, S.B \leq 4.9, S.C = T.C, 1 \leq E \leq 5\}$

The reader should note here, that the MDR refinements are also present in the set of refinements FPE. Indeed, the root node of every graph $G_{Q'}$ corresponds to the MDR query Q'. Thus, the set $\mathcal{Q}'_{mdr}$ is the final set of query refinements as

$$\mathcal{Q}'_{mdr} \subseteq \mathcal{Q}'_{fpe}$$

Example 5.4.8. *From the graphs $G_{Q_{t_1}}$, $G_{Q_{t_2}}$, $G_{Q_{t_3}}$ we obtain the set of refined queries $\mathcal{Q}'_{fpe}$ displayed in Table 5.4.2.*

5.4.3 Joins-Only explanations

The second type of explanations encountered in a Why-Not answer polynomial is about explanations with join conditions only. Here, the previous approach of condition relaxation is not applicable. When a join exists in an explanation, this means that for some compatible tuples no join partners were found. One obvious solution to this problem would be to change the join to cross product. However, this trivial solution may generate numerous false positive tuples would be overwhelming. Thus, the refinement may not be useful for the user nor meet her initial intent. As a different option, we resort to left and right outer joins as already mentioned in the introduction.As we discuss in this section, it is not always feasible to refine the query using this approach. So, it is not guaranteed that we will obtain a query refinement for every explanation including joins. Note that to fill this gap, in the future we will consider different techniques for refinements, like for example using foreign keys, as discussed later on in the Perspectives section (Chapter 6).

Our approach for processing joins-only explanations is totally different from the case of selections-only., as it does not rely on compatible tuples. Moreover, here we do not proceed to a false positive elimination phase, for which the compatible tuples were previously used. Nevertheless, for joins-only we leave the false positive elimination phase as an open question.

Let us now describe why and when a solution involving left and right outer joins is feasible. Intuitively, this depends on the direct and indirect compatible tuples (recall that these notions have been introduced for *NedExplain*, Section 4.1.3), and how the relations (or partitions) storing direct and indirect tuples, connect with each other through the joins of the query. To ease the discussion, we blaze the trail using examples/use cases, based on the database in Figure 5.6.

In the first two examples, the queries involve only join conditions. In the third example we introduce selections in the query, however the reader should remember that the explanation is still composed only by join conditions.

Book

Title	BAuthor	Price	Pub	B_Id
Odyssey	Homer	15	Cambridge	Id$_1$
Iliad	Homer	45	Prestwick	Id$_2$
Antigone	Sophocles	49	Psichogios	Id$_3$
Lysistrata	Aristophanes	30	Hackett	Id$_4$

Author

Name	Dob	A_Id
Homer	800BC	Id$_5$
Sophocles	400BC	Id$_6$
Euripides	400BC	Id$_7$
Aristophanes	400BC	Id$_8$

Publisher

Appellation	Country	P_Id
Cambridge	England	Id$_9$
Prestwick	USA	Id$_{10}$
Psichogios	Greece	Id$_{11}$

Figure 5.6: Sample database for the case study in joins-only explanations.

Example 5.4.9. ***Explanation condition over one direct and one indirect relation***
Consider the following scenario over the tables Author and Book of Figure 5.6, where for convenience the attributes are displayed only by their name.

Query *Q:*	$(\{Author, Book\}, \{Name, Title\},$ $\{Name = BAuthor\})$
Query result $Q[\mathcal{I}]$*:*	$\{(Homer, Odyssey), (Homer, Iliad),$ $(Sophocles, Antigone), (Aristophanes, Lysistrata)\}$
Why-Not question *WN:*	$\{Name = Euripides\}$
Explanation $\mathcal{E}$*:*	$Name = BAuthor$

Since the Why-Not question is specified over the Author relation, the Author is a direct relation, whereas the Book is an indirect relation. This means that we absolutely want the values from the (partial) compatible tuples from Author to appear in the result tuples. The joins-only explanation $Name = BAuthor$ *informs us that there were no join partner tuples in the Book relation for the author Euripides. So, in order to be able to obtain Euripides*

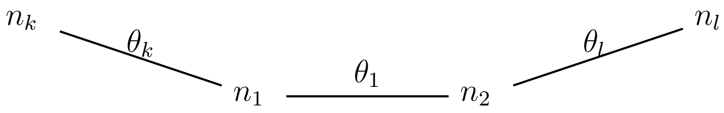


Figure 5.7: Query Q graph, where nodes n denote schema relations and edges θ denote joins.

in the result, we can change the join to a left outer join, with the direct relation in the left side of the operator to ensure that Euripides will appear in the result. The refined query Q' then, has the condition $Name ⟕_{=} BAuthor$ instead of $Name = BAuthor$ and can be expressed in SQL as:

```
SELECT Author.Name,Book.Title
FROM Author
LEFT OUTER JOIN
Book ON Author.Name = Book.BAuthor
```

Indeed, the result of Q' contains the tuple $(Euripides, null)$ along with the original result tuples.

In the previous example, we demonstrate the simplest case that can be treated in this category: *the explanation contains only one join that is over one direct (R_{Dir}) and one indirect (R_{InDir}) relation, whereas there are no other conditions in the query.* In this case, the refined query Q' can be obtained from Q by the following transformation:

$$R_{Dir}.A\ \theta\ R_{InDir}.A' \quad \Rightarrow \quad R_{Dir}.A ⟕_{\theta} R_{InDir}.A'$$

More generally, consider a query with several joins, represented as the graph in Figure 5.4.3, where a node n stands for a schema relation and an edge θ stands for a join among two relations. Consider that n_1 corresponds to R_{Dir} and n_2 to R_{InDir}. If the join θ_1 forms a joins-only explanation and if there is no direct relation reachable from n_1 through n_2, then the refinement using (left or right) outer joins **can be applied**. For example if θ_l models a 'transitive' join, then n_l should not be a direct relation.

Example 5.4.10. ***Explanation condition over two direct relations***

Consider a variation of the previous example, with the Why-Not question being specified over both relations, making them both direct.

Query *Q:*	$(\{Author, Book\}, \{Name, Title\},$ $\{Name = BAuthor\})$
Query result *$Q[\mathcal{I}]$:*	$\{(Homer, Odyssey), (Homer, Iliad),$ $(Sophocles, Antigone), (Aristophanes, Lysistrata)\}$
Why-Not question *WN:*	$\{Name = Euripides, Title = Odyssey\}$
Explanation *$\mathcal{E}$:*	$Name = BAuthor$

In this case, the user expects to find a result tuple with both the values Euripides and Odyssey. However, this is impossible to achieve by turning the join to an outer join (left, right or full); what we can achieve is obtaining some tuples with $Name =$ '$Euripides'$ and some other with $Title =$ '$Odyssey'$, which is not what is expected.

The previous example demonstrates a simple case where the solution with outer joins is not applicable, i.e., when the explanation contains one join that is specified over two direct relations.

More generally, consider again the graph in Figure 5.4.3. If θ_1 forms a joins-only explanation and both n_1 and n_2 are direct relations, then the refinement using (left or right) **cannot be applied**.

Example 5.4.11. ***Explanation conditions on indirect relations***

Consider the following scenario

Query Q*:*	$(\{Author, Book, Publisher\}, \{Name, Country\}, \{Name = BAuthor, Pub = Appellation\})$
Query result $Q[\mathcal{I}]$*:*	$\{(Homer, England), (Homer, USA), (Sophocles, Greece)\}$
Why-Not question WN*:*	$\{Name = Aristophanes\}$
Explanation $\mathcal{E}$*:*	$Pub = Appellation$

Now, the query involves two joins, the first one joining the direct relation Author with the indirect relation Book and the second one joining the indirect Book with the indirect relation Publisher. The second join (Pub = Appellation) is the one identified by the explanation as causing the problem, and thus should be fixed. Since no direct relations follow the indirect ones in the join chain, we can fix this join by turning it into a left outer join.

So, we obtain the query Q' *(in SQL):*

```
SELECT Author.Name,Publisher.Country
FROM Author
INNER JOIN
Book ON Author.Name = Book.BAuthor
LEFT OUTER JOIN
Publisher ON Book.Pub = Publisher.Appellation
```

So, indeed in the result of Q' *we obtain* $(Aristophanes, null)$ *along with the original result tuples.*

The previous example demonstrates the case where the explanation condition is specified over indirect relations and which we can indeed treat with the left outer join solution.

More generally, consider again the graph in Figure 5.4.3. If θ_1 forms a joins-only explanation, both n_1 and n_2 are indirect relations and there does not exist a path connecting two direct relations and going through the join θ_1, then the refinement using (left or right) outer join **can be applied**. This means that the nodes n_k and n_l cannot be direct relations.

Then, fixing the join in Q to a left outer join in Q' consists in

$$R_{InDir}.A\ \theta\ R'_{InDir}.A \quad \Rightarrow \quad R_{InDir}.A ⟕_{\theta} R'_{InDir}.A$$

Note that in all the examples, the direction of the outer join, left or right, is determined by the side on which the direct relation reside.

The three previous examples lead to one common definition for query refinements using right or left outer join after stating when such a refinement is valid.

Definition 5.4.5. *(Joins-only Refinement)*

Let $Q{=}(\mathcal{S}, \Gamma, C)$ *be a query. Let* $\mathcal{E}{=}\{op_1\}$ *be a joins-only explanation, where* op_1 : $\{R_1.A\ \theta\ R_2.A\} \in C$. *Let* G_Q *denote the graph modelling the relations (nodes R) and joins (edges op) of the query* Q.
Given $\mathcal{E}$, *a joins-only refinement for* Q *is feasible, iff*

> *for each path* Π *in* G_Q *from relation* R_k *to* R_l, *where* R_k, R_l *are direct,* ***it holds that*** op_1 *is not an edge in* Π

A joins-only refinement Q' *for* Q *is defined by*

$$Q'{=}(\mathcal{S}_Q, \Gamma_Q, C')$$

where

$$C'{=}(C \setminus \{op_1\}) \cup \{R_1 \leftouterjoin_{\theta} R_2\}$$

if
— R_1 *is a direct relation, or*
— *there exists a path* Π *from relation* R_1 *to* R_k, *where* R_k *is direct, and* R_2 *is not a node in* Π.

or

$$C'{=}(C \setminus \{op_1\}) \cup \{R_1 \rightouterjoin_{\theta} R_2\}$$

if
— R_2 *is a direct relation, or*
— *there exists a path* Π *from relation* R_2 *to* R_k, *where* R_k *is direct, and* R_1 *is not a node in* Π.

The previous definition can be easily extended for the case of multiple joins in the explanation $\mathcal{E}$.

Up to now, we considered a query Q consisting of only join conditions. This facilitated our discussion to determine the shape of the query graph in correlation with the join conditions in the explanation, in order to be able to propose joins-only query refinements. We also showed how we choose the direction of the outer-join to be used, depending on the location of the direct relations.

Now, we continue with the example with a query containing selection conditions, besides the joins assuming that a joins-only refinement is applicable for the query Q, given the explanation $\mathcal{E}$. As a reminder, the selections do not appear in the explanation as we are only considering here joins-only explanations.

Example 5.4.12. ***Query with selections***

Assume that the tuple

Antigone	*Sophocles*	*49*	*Psichogios*	*Id*$_3$

in the Books instance changes to:

Antigone	*Sophocles*	*null*	*Psichogios*	*Id*$_3$

Now, consider the following scenario:

Query Q*:*	$(\{Author, Book\}, \{Name, Title\},$ $\{Name = BAuthor, Price > 30\})$
Query result $Q[\mathcal{I}]$*:*	$\{(Homer, Iliad)\}$
Why-Not question WN*:*	$\{Name = Euripides\}$
Explanation $\mathcal{E}$*:*	$Name = BAuthor$

This is the same scenario as in our first example, with an extra selection condition for the price of the retrieved books. In this case, we create a sub-query to retrieve the books with price greater than 30, and perform the left outer join on the result of the subquery and the Author relation. So, the refined query Q' *is (in SQL):*

```
SELECT Author.Name, sub.Title
FROM Author
LEFT OUTER JOIN
(SELECT Book.Title,Book.BAuthor FROM Book WHERE Book.Price>30) sub
ON Author.Name = sub.BAuthor
```

So, indeed in the result we obtain the tuple $(Euripides, null)$ *along with the original result tuple.*

In this example, the query and explanation are formed in such a way that we can apply the outer-join solution. However, the query conditions contain also selections. In this case we add one more step to the solution adding the selections in the refined query.

This extra step creates subqueries replacing relations over which selections are specified. Then, the joins of the query are specified over the respective subqueries in the refined query.

This approach is justified as follows: keeping the selection conditions in the WHERE clause of the refined query Q', would prune out of the result any tuple with a *Null* value in the respective attributes. Thus, the WHERE clause would also prune out the *Null*-padded tuples generated by the OUTER JOIN that we need to create the missing tuples.

Note that introducing subqueries in the query, yields an output schema $\Gamma_{Q'}$ for the refined query Q' that is different from the output schema Γ_Q of Q. This causes an inconsistency w.r.t. our general definition of query refinement (Definition 5.3.1, page 122). There, to verify that a query refinement recovers some missing tuples, we assumed that the output schema of Q' complies with the schema of the compatible tuples, i.e., $\mathcal{S}_Q$.

However, this inconsistency can be easily overpassed. Each introduced subquery *sub* refers to a set of relations $\mathcal{R}_{sub}$. Consider the case when the attributes of the relations of $\mathcal{R}_{sub}$ are projected out by the query Q, i.e., $\mathcal{A}(\mathcal{R}_{sub}) \subseteq \Gamma_Q$. Then, since each relation appears only in one subquery, there is an injective non-surjective mapping μ from attributes in $\Gamma_{Q'}$ to attributes in $\mathcal{S}_Q$ (each attribute in $\Gamma_{Q'}$ maps to exactly one attribute in $\mathcal{S}_Q$, and no other attribute from $\Gamma_{Q'}$ maps to the same attribute from $\mathcal{S}_Q$). Consequently, Definition 5.3.1 can be stated as:

'A query refinement is a query Q' s.t. $\exists t \in CT :\ t \models \mathcal{E}$ and $\pi_\Gamma[t] \in \mu(Q'[\mathcal{I}])$. '

One other possible approach to integrate the selections in the query Q', would be to use the predicate *is Null*, instead of introducing subqueries. Following this approach, we could disjunctively add in the WHERE clause the predicate A *is Null*, where A is an attribute over which a selection is specified and also is in the schema of a relation in the right relation if a LEFT OUTER JOIN is used (or the left relation if a RIGHT OUTER JOIN is used).

However, adding the predicate A *is Null* may also introduce false positive tuples, computed from source tuples having an A Null value. This would increase the number of false positive tuples in the refined query result. Thus, we have opted for the first solution using subqueries. Nevertheless, an optimized version of our algorithm would exploit schema constraints to find attributes without *Null* values and in this case choose the refinement without subqueries.

For example, in the example of this case the refined query would be

```
SELECT Author.Name, Book.Title
FROM Author
LEFT OUTER JOIN
 Book
ON Author.Name = Book.BAuthor
WHERE Book.Price>30 OR Book.Price is Null
```

This query would return the result $\{(Homer, Iliad), (Euripides, null), (Sophocles, null)\}$. Indeed, the *is Null* condition has allowed the false positive tuple $(Sophocles, null)$ appear in the result. However, this tuple was pruned out with the *subqueries* approach, providing a better result.

5.4.4 Mixed explanations

The third case that we may have to deal with, is when both selection and join conditions occur in an explanation of the Why-Not answer polynomial. Since the solution is directly combining the two previous cases, we do not elaborate further details on this.

5.5 FixTed Algorithm

In this section, we provide the *FixTed* algorithm that leverages Why-Not answer polynomials in order to generate query refinements as defined in Definition 5.3.1, page 122. Briefly, *FixTed* computes query refinements as per explanation type and returns only the most interesting (regarding similarity and precision) ones using a cost function that ranks the refinements based on a number of metrics. Next, we describe in detail the algorithm, outlining the associated pseudo-code.

Algorithm 9 describes the main steps of *FixTed*. *FixTed* accepts as input a conjunctive query with inequalities Q and a precomputed Why-Not answer polynomial *PEX* w.r.t. some Why-Not question. Also, it requires as input the set $\mathcal{V}$ of compatible tuples organised in partition views V_i as already discussed in Section 2.2. $\mathcal{V}$ and *PEX* are output by *Ted++* (see Algorithm 7). The input also includes the weights $\mathcal{W}$ for the metrics used in the query refinement ranking function. The metrics and the ranking function are described later on.

First, in lines 1-2 we group the explanations (i.e., the terms) from the polynomial w.r.t.

```
Algorithm 9: FixTed
   Input: $Q$: query, $\mathcal{V}$: set of partial compatible tuples views w.r.t. a Why-Not
          question and the valid partitioning $\mathcal{P}$, $PEX$: Why-Not answer polynomial,
          $\mathcal{W}$: weights for scoring function
   Output: $\mathcal{Q}'$: set of query refinements for $Q$
 1 $W_1 \leftarrow \{\mathcal{E} \mid \mathcal{E} \in PEX \; and \; \forall op \in \mathcal{E} : op$ is a selection condition$\}$;
 2 $W_2 \leftarrow \{\mathcal{E} \mid \mathcal{E} \in PEX \; and \; \forall op \in \mathcal{E} : op$ is a join condition$\}$;
 3 $W_3 \leftarrow PEX \setminus (W_1 \cup W_2)$;
 4 $\mathcal{Q}'_{mdr} \leftarrow$ MDR$(Q, W_1, CT, \mathcal{P})$; % minimum distance refinement, for selections-only
   explanations%
 5 $\mathcal{Q}'_{fpe} \leftarrow$ FPE$(Q, \mathcal{Q}'_{mdr})$; % false positive tuples elimination, for selections-only
   explanations%
 6 $\mathcal{Q}'_{jo} \leftarrow$ JoinsOnly$(Q, W_2)$; % joins-only explanations%
 7 $\mathcal{Q}'_{mixed} \leftarrow$ Mixed$(Q, W_3, CT)$; % mixed explanations%
 8 $\mathcal{Q}' \leftarrow \mathcal{Q}'_{mdr} \cup \mathcal{Q}'_{fpe} \cup \mathcal{Q}'_{jo} \cup \mathcal{Q}'_{mixed}$;
 9 for refined query $Q' \in \mathcal{Q}'$ do
10     $Q'.nfp \leftarrow computeNumberOfFalsePositiveTuples(Q')$;
11     $Q'.vd \leftarrow computeValueDistance(Q')$;
12     $Q'.ncc \leftarrow computeNumberOfChangedConditions(Q')$;
13     $Q'.nac \leftarrow computeNumberOfAddedConditions(Q')$;
14 $\mathcal{Q}' \leftarrow \mathcal{SL}(\{fp, vd, nac, ncc\}, \mathcal{Q}')$;
15 for refined query $Q' \in \mathcal{Q}'$ do
16     $Q'.score \leftarrow assignScore(Q', \vec{\mathcal{W}})$; % assign score based on scoring function
       weights%
17 return rank($\mathcal{Q}'$);
```

their type. Then, to compute the query refinements of Q based on the explanations, we follow a different procedure for each group.

Selections-only explanation set W_1 (Algorithm 9 lines 4 & 5) For the selections-only explanations we perform two consecutive steps, as described in Section 5.4.1. We start with the *Minimum Distance Refinements(MDR)* phase, sketched in Algorithm 10, and continue with the *False Positive tuples Elimination (FPE)* phase, sketched in Algorithm 11.

Algorithm 10 iterates through the explanations in W_1 starting from the explanations containing the least number of conditions and moving on in ascending order. For each explanation $\mathcal{E}$ we firstly compute the partial compatible tuples eliminated by the explanation (line 4). If $Part_{\mathcal{E}}$ is the set of partitions over which the conditions in $\mathcal{E}$ is specified, then the view storing the partial compatible eliminated tuples by $\mathcal{E}$ is specified by the statement:

$$V_{\mathcal{E}}=\sigma_{Pred}[\bigtimes_{R \in Part \in Part_{\mathcal{E}}} R]$$

where $Pred = \bigwedge_{op \in \mathcal{E}} \neg op \qquad \bigwedge_{op \in (C_Q \setminus \mathcal{E}) \text{ s.t. } \mathcal{A}(op) \subseteq \mathcal{A}(Part_{\mathcal{E}})} op.$

Algorithm 10: MDR Algorithm

```
   Input: $Q$: query, $W_1$: set of explanations with selection conditions only, $\mathcal{V}$: set of
          partial compatible tuples views, $\mathcal{P}$ : valid partitioning of $\mathcal{S}_Q$
   Output: $\mathcal{Q'}_{mdr}$: set of minimum distance query refinements
1  $\mathcal{Q'}_{mdr} \leftarrow \emptyset$;
2  for $explanation\ \mathcal{E} \in W_1$ % accessed in ascending explanation size order% do
3      $Part_{\mathcal{E}} \leftarrow \{Part \mid Part \in \mathcal{P}\ and\ \exists R \in Part\ s.t.\ \mathcal{E}$ is specified over $R\}$;
4      $V_{\mathcal{E}} \leftarrow$ partial compatible tuples eliminated by $\mathcal{E}$;
5      $V_{sub}^{SL} \leftarrow \{V_{\mathcal{E}'}^{SL} \mid \mathcal{E}' \in W_1\ and\ \mathcal{E}' \subset \mathcal{E}\}$;
6      $V_{\mathcal{E}}^{SL} \leftarrow localSkylineTuples(V_{\mathcal{E}}, \mathcal{A}(\mathcal{E}), V_{sub}^{SL})$; % Definition 5.4.1%
7      $V_{\mathcal{E}}^{dSL} \leftarrow dominantSkylineTuples(V_{\mathcal{E}}^{SL}, \mathcal{A}(Part_{\mathcal{E}}) \setminus \mathcal{A}(\mathcal{E}))$;
8      for $t \in V_{\mathcal{E}}^{dSL}$ do
9          $Q' \leftarrow$refineQueryMDR$(Q, \mathcal{E}, t)$;
10         $\mathcal{Q'}_{mdr} \leftarrow \mathcal{Q'}_{mdr} \cup \{Q'\}$;
11 return $\mathcal{Q'}_{mdr}$
```

Intuitively, $V_{\mathcal{E}}$ computes the tuples in the partitions of $\mathcal{E}$ that do not satisfy the conditions of the explanation but satisfy all other query conditions specified over the attributes of these partitions. In this way we know that a repair of the conditions in $\mathcal{E}$ is sufficient to allow some compatible tuples (from $V_{\mathcal{E}}$) to make it to the result.

Example 5.5.1. *Let us revisit Example 5.3.1, on page 123 and focus on the set of selections-only explanations set W_1. All explanations in W_1 contain selections over the relation R, which as we said forms the partition $Part_1$. So, all the views $V_{\mathcal{E}}$ have the same schema as R, i.e., $\{A, B, C\}$, and are subsets of Table 5.1. As, each compatible tuple t in Table 5.1 has a distinct result $\pi_{A,B,C}[\{t\}]$, we abusively refer to the partial tuple $t_{|R}$ as t. Note that this would not be the case if more than one compatible tuples led to the same partial tuple.*

It is clear that the views $V_{\mathcal{E}}$ correspond to compatible tuples marked with t in each quarter of Figure 5.2(b). For instance, $V_{\{A<5.5\}}$=$\{t_2, t_8, t_9\}$ corresponds to the fourth quarter.

Then, we compute the set of local skyline tuples, using also the set of local skyline tuples of sub-explanations of $\mathcal{E}$ (lines 5 & 6). Note here, that we can compute local skyline *partial* compatible tuples instead of full compatible tuples, because the attributes of the explanation (dimensions for the skyline) are available in the involved partitions.

Computing skyline tuples is equivalent to the problem of computing the skyline over a set of n points given d dimensions. If we consider a naive algorithm (nested-loop algorithm) to compute the skyline tuples, where each point $p \in P$ is compared with each other point $p' \in P$ in all n dimensions, the worst time complexity is $O(n^2)$. If we consider a more clever algorithm like the Divide and Conquer (D&C), the complexity is $O(nlogn)$ for d=2,3 and $O(n^{[d/2]+1})$ for $d > 3$ [BKS01], where d is the dimensionality and n the cardinality of the input set. No matter the algorithm we choose, the skyline computation heavily depends on the cardinality of the set of compatible tuples set and the number of dimensions. Note, that if we did not have the explanations available, we would have considered $\mid CT \mid$ tuples and $\mid \mathcal{A}(W_1) \mid$ dimensions in the worst case. The fact that we are considering local skyline tuples provides an opportunity for more efficient process because we consider

— $\mid \pi_{\mathcal{A}(Part_{\mathcal{E}})}\sigma_{C'}[CT] \mid$ number of compatible tuples, instead of $\mid CT \mid$, where $Part_{\mathcal{E}}$ is the set of partitions over which $\mathcal{E}$ is defined, C' is a predicate obtained considering the explanation $\mathcal{E}$ and $\sigma_{C'}[CT]$ are the compatible tuples eliminated only by $\mathcal{E}$, and

— $\mid \mathcal{A}(\mathcal{E}) \mid$ number of considered attributes, instead of $\mid \mathcal{A}(C) \mid$, as dimensions.

Example 5.5.2. *In Example 5.4.1, on page 5.4.1 we already gave the skyline tuples for each explanation, marked in Figure 5.2(b).*

Lets us now discuss how considering local skyline tuples could be more efficient than computing skyline tuples over all the attributes constrained in the query Q. If we were to find the skyline tuples in the set of full compatible tuples t_1, ..., t_{11}, we would have to make at most two comparisons (one for each attribute A, B) for each pair of compatible tuples. If we take into account the formula

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

computing the combinations of size k of a set of n elements [Hal98]. This yields a worst-case number of comparisons equal to

$$\chi_{sk}{=}2 * \frac{11!}{2!9!} = 110$$

On the other hand, let us see how many comparisons we do with the local skyline tuples.

For the explanation $\{A < 5.5\}$ there are 3 tuples to be compared on one attribute (A), for the explanation $\{B < 4\}$ we have 4 tuples to be compared on 1 attribute (B) and for $\{A < 5.5, B < 4\}$ we have 4 tuples to be compared on 2 attributes (A, B).

Thus the number of comparisons is

$$\chi'_{lsk} = 1 * \frac{3!}{2!1!} + 1 * \frac{4!}{2!1!} + 2 * \frac{4!}{2!2!} = 3 + 12 + 12 = 27$$

*To this number we add the extra comparisons for the sub-combinations needed for $\{A < 5.5, B < 4\}$, which sum up to 3*4 comparisons on 1 attribute (1 skyline tuple of $\{A < 5.5\}$ plus 2 skyline tuples of $\{B < 4\}$ compared with every compatible tuple associated with $\{A < 5.5, B < 4\}$). Thus, in the worst case the number of needed comparisons is*

$$\chi_{lsk} = 27 + 12{=}39$$

As a result, $\chi_{lsk} < \chi_{sk}$. Moreover, in our example we have achieved to reduce the number of comparisons per 71, performing a little more than one third of the comparisons needed for the standard skyline computation. Note that the number of comparisons may decrease even more substantially, in the case when the number of partial tuples is less than the number of full tuples, i.e., in the case when more than one tuples are projected to the same tuple, considering the schema of one partition.

Line 7 prunes the space of local skyline tuples to keep only dominant skyline tuples. As here we handle partial compatible tuples, the dominant skyline tuples are computed based on the attributes of the involved partitions. So, in a set of local skyline tuples, if there are tuples that are equal in the dimensions specified by the attributes of $\mathcal{E}$, we only consider those that are not dominated by the others based on the remaining attributes of the involved partitions. Note here, that we do not consider all the remaining attributes

Algorithm 11: FPE Algorithm

```
   Input: $\mathcal{Q}'_{mdr}$: set of refined queries, each one associated with one explanation and
          one tuple, $Res$ the query result before the final projection
   Output: $\mathcal{Q}'_{fpe}$: set of query refinements
1  for query $Q' \in \mathcal{Q}'_{mdr}$ do
2      $\mathcal{E} \leftarrow$ explanation associated with $Q'$;
3      $t \leftarrow$ tuple associated with $Q'$;
4      $Part_{\mathcal{E}} \leftarrow$ set of partitions over which the explanation $\mathcal{E}$ is specified;
5      $\mathcal{C} \leftarrow \emptyset$; %new conditions%
6      for attribute $A \in (\mathcal{A}(Part_{\mathcal{E}}) \setminus \mathcal{A}(\mathcal{E}))$ do
7          if $A \in \mathcal{A}(W_1)$ then
8              $min_A \leftarrow min(t.A, min(\pi_A(Res)))$; $\mathcal{C} \leftarrow \mathcal{C} \cup min_A \leq A$;
9          else
10             $min_A \leftarrow min(t.A, min(\pi_A(Res)))$;
11             $max_A \leftarrow min(t.A, max(\pi_A(Res)))$;
12             $\mathcal{C} \leftarrow \mathcal{C} \cup min_A \leq A \leq max_A$;
13     $ZeroFP \leftarrow \emptyset$; %set of condition sets that lead to a query with zero false
       positive tuples%
14     for condition set condComb in $2^{\mathcal{C}} \setminus \emptyset$ do
15         if $\nexists condComb' \in ZeroFP$ s.t. $condComb' \subseteq condComb$ then
16             $C_{fpe} \leftarrow C_{Q'} \cup condComb$;
17             $Q' \leftarrow (S_Q, \Gamma_Q, C_{fpe})$;
18             $\mathcal{Q}'_{fpe} \leftarrow \mathcal{Q}'_{fpe} \cup Q'$;
19             if $Q'[\mathcal{I}] \setminus (Q[\mathcal{I}] \cup \pi_{\Gamma_Q}[CT]) == \emptyset$ then
20                 $ZeroFP \leftarrow ZeroFP \cup \{condComb\}$;
21 return $\mathcal{Q}'_{fpe}$
```

of the schema $\mathcal{S}_Q$, as in Definition 5.4.3 because they are not available in the schema of $V_{\mathcal{E}}$. However, this does not cause a problem during the FPE refining phase, as will be explained shortly after.

Finally, we use every dominant local skyline tuple to change the respective (to the explanation) query conditions and create a refined query per tuple, added in $\mathcal{Q}'_{mdr}$.

Algorithm 11 gradually eliminates false positive tuples from the refined queries in $\mathcal{Q}'_{mdr}$. Each refined query $Q' \in \mathcal{Q}'_{mdr}$, is associated with

1. an explanation $\mathcal{E}$,
2. a dominant local skyline (partial) tuple t, and
3. with the set of partitions of $\mathcal{E}$

We are interested in the attributes from the partitions that are not constrained by $\mathcal{E}$ and based on which we introduce a new condition. To construct the new conditions per attribute (Algorithm 11,lines 6-12) we follow the process explained in Section 5.4.2, page 132.

Note however a slight difference, compared to what is described in Section 5.4.2. Instead of considering all the attributes of the query input schema not constrained by $\mathcal{E}$, we only consider the ones that are in the partitions of $\mathcal{E}$. This follows the remark we made for the computation of the dominant skyline tuples, in Algorithm 10, line 7. In this way we prefer to avoid the complexity of computing full compatible tuples, at the price of missing some more query refinements that would possibly eliminate false positive tuples.

Example 5.5.3. *For the dominant skyline tuples we have seen in Example 5.4.4, on page 133 that we took into consideration the attributes from partition* $Part_2$ *as well. Here, we are considering only the attributes A, B, C from the partition* $Part_1$*. In our example the resulting dominant tuple sets are the same as with in Example 5.4.4.*

To clarify a different case, consider for the explanation $\mathcal{E}_2 = \{B < 4\}$ *that the set of local skyline tuples contains also* t'_1=$(R.A : 1.6, R.B : 5.6, 3, 0)$*, i.e,* $lSL_{\mathcal{E}_2}$=$\{t_1, t'_1, t_6\}$*. Then, by Definition 5.4.3 (taking into account the attributes* A, C, D, E*) the set of dominant skyline tuples for* $\mathcal{E}_2$ *is* $dSL'_{\mathcal{E}_2}$=$\{t_1, t'_1\}$*. If we consider however only the attributes from* $Part_1$ *we have* $dSL_{\mathcal{E}_2}$=$\{t_1\}$*.*

Thus, we would miss the dominant skyline tuple t'_1 *based on which we could add some more refinements in the FPE phase, and possibly improve the precision. Nevertheless, we gain on execution time both in this phase and in the FPE phase, where the search space becomes smaller.*

As a reminder, all the possible query refinements are modelled as a graph like in Figure 5.3. Lines 14 - 18 correspond to building a query for each node in the graph, i.e., each element from the powerset of the new conditions set.

As already said, when a node in the graph yields zero false positive tuples, then the children nodes are not considered. Line 19 checks if a refined query (i.e., the addition of a specific condition combination) does not produces false positive tuples. In this case, we memorize it in order for the next loops to skip all supersets of this condition combination. Finally, all refined queries obtained in this phase are returned in the set $\mathcal{Q}'_{fpe}$.

Example 5.5.4. *In Example 5.4.7 page 138, we created the graphs associated with each dominant skyline tuple. Based on these graphs we obtained the refinements in Table 5.4.2.*

As in the algorithm we are considering only the attributes in the partitions associated with each explanation, the graphs that we build for each dominant skyline tuple are subgraphs of the ones in Example 5.4.7. More specifically, the nodes whose label contains the attributes D or E are not included (neither the edges coming in or out of these nodes). As a consequence, the query refinements obtained by FixTed *based on the explanation in set* W_1 *are displayed in Table 5.3, rows* $\mathcal{Q}'_{mdr}$ *and* $\mathcal{Q}'_{fpe}$*, and are a subset of Table 5.4.2.*

Joins-only explanation set W_2 and mixed explanations set W_3 (Algorithm 9 lines 6 & 7) For the joins-only explanations, *FixTed* implements the case study discussed in Section 5.4.3. Thus, if the scenario meets the requirements of Definition 5.4.5, one query refinement is generated per explanation in W_2.

For the mixed explanations, the solution is a combination of the previous as discussed in Section 5.4.4. Thus, if the scenario meets the requirements, a set of query refinements is generated per explanation in W_3.

If the scenario associated with an explanation $\mathcal{E}$ does not meet the requirements then no refinement is returned for $\mathcal{E}$.

Table 5.3: Refined queries for scenario of Example 5.3.1 using *FixTed*.

$\mathcal{Q}'_{mdr}$	Q'_{t_1}=$(\mathcal{S}_Q, \Gamma_Q, \{R.A < 5.5, S.B \leq 5.6, S.C = T.C\})$
	Q'_{t_2}=$(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 7.5, S.B < 4, S.C = T.C\})$
	Q'_{t_3}=$(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 6.4, S.B \leq 4.9, S.C = T.C\})$
$\mathcal{Q}'_{fpe}$	$Q''_{t_1,\{A\}}$=$(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 2.1, S.B \leq 5.6, S.C = T.C\})$
	$Q''_{t_1,\{C\}}$=$(\mathcal{S}_Q, \Gamma_Q, \{R.A < 5.5, S.B \leq 5.6, S.C = T.C, 2.8 \leq C \leq 7.5\})$
	$Q''_{t_2,\{B\}}$=$(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 7.5, S.B \leq 2.4, S.C = T.C\})$
	$Q''_{t_2,\{C\}}$=$(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 7.5, S.B < 4, S.C = T.C, 2.8 \leq C \leq 8.4\})$
	$Q''_{t_3,\{C\}}$=$(\mathcal{S}_Q, \Gamma_Q, \{R.A \leq 6.4, S.B \leq 4.9, S.C = T.C, 2.8 \leq C \leq 7.5\})$
$\mathcal{Q}'_{jo}$	Q'=$(\mathcal{S}', \mathcal{A}(\mathcal{S}'), \{sub_R \ltimes_{sub_R.C=S.C} S\})$
	where $\mathcal{S}' = (\mathcal{S} \setminus \{R\}) \cup sub_R$, and
	sub_R=$(\{R\}, \mathcal{A}(R), \{R.A < 5.5, R.B < 4\})$

Example 5.5.5. *The joins-only explanation of the running example is composed by* op_3=$\{R.C = S.C\}$*. As we said in the description of the scenario, the relation* R *is indirect and* S *is direct. So, the 'picky' join is joining one direct relation with one indirect, whereas there are no other direct relations. Thus, based on Definition5.4.5, a joins-only refinement is feasible for* Q *and* $\{op_3\}$*.*

The condition set C_Q *contains two selections over the relation* R*. So, we create the following subquery which will be used in the place of* R *in the refinement*

$$sub_R=(\{R\}, \mathcal{A}(R), \{R.A < 5.5, R.B < 4\})$$

The joins-only query refinement is then:

$$Q'=(\mathcal{S}', \mathcal{A}(\mathcal{S}'), \{sub_R \ltimes_{sub_R.C=S.C} S\})$$

where $\mathcal{S}' = (\mathcal{S} \setminus \{R\}) \cup sub_R$*.*

It is obvious now that the compatible tuple t_{12}=$(R.A : 2, R.B : 2, R.C : 3, S.C : 4, S.D : 6, S.E : 5)$ *satisfies the conditions of the refinement and so is included in the result.*

All query refinements are now listed in Table 5.3.

Back to Algorithm 9, in lines 10 - 13 we compute a number of values for each refined query. These values correspond to the metrics used for comparing, pruning and ranking the refined queries. In more detail, the metrics are:

1. *ncc*: number of changed conditions
2. *nac*: number of added conditions
3. *vd*: total difference in the changed values when numerical values, or edit distance if strings
4. *nfp*: number of false positive tuples

Note that for refinements obtained by joins-only explanation we assume that $vd = 0$, since in that case the operator (join) has been changed, but not constants were involved.

Table 5.4: Refined queries by *FixTed*, with metrics. Scores are assigned only to skyline queries, which are returned to the user in the order appearing in the table.

	Query	**ncc**	**nac**	**vd**	**nfp**	**score**
Q'	Q'_{t_1}	1	0	$5.6 - 4 = 1.6$	1	2.56
	Q'_{jo}	1	0	0	2	2.5
	$Q''_{t_1,\{C\}}$	1	2	1.6	0	2.06
	$Q''_{t_2,\{B\}}$	2	0	$2 + 4 - 2.4 = 3.6$	0	1
	Q'_{t_2}	1	0	$7.5 - 5.5 = 2$	1	
	Q'_{t_3}	2	0	$6.4 - 5.5 + 4.9 - 4 = 1.8$	2	
	$Q''_{t_1,\{A\}}$	2	0	$1.6 + 5.5 - 2.1 = 5$	0	
	$Q''_{t_2,\{C\}}$	1	2	2	0	
	$Q''_{t_3,\{C\}}$	2	2	1.8	0	

FixTed returns only the skyline refined queries computed using the metrics as dimensions (line 14). Note that this is a new skyline that we define over the set of refined queries using four dimensions (corresponding to the four metrics previously mentioned). Here, the least values are preferable, in order to minimize the difference and the false positive tuples of the refined queries.

Moreover, in line 16, for each refined query we calculate its score using the cost function

$$score = \beta_{ncc} ncc_o + \beta_{nac} nac_o + \beta_{vd} vd_o + \beta_{nfp} nfp_o$$

where $\beta_x \in [0, 1]$ for $x \in \{nwc, nac, vd, nfp\}$ is the corresponding metric weight from $(\vec{\mathcal{W}})$.

Also, x_o stands for the normalised value of the metric x. If x_{max} is the maximum value for the metric x over all the refinements, then x_o is defined as follows

$$x_o = 1 - \frac{x}{x_{max}}$$

This score is used in order to rank the refined queries and provide an ordering over them.

In the future we plan to investigate more sophisticated metrics and ranking functions.

Example 5.5.6. *Table 5.4 displays for each refined query from Table 5.3, the values for the metrics used for pruning and ranking.*

Based on the metrics we also compute the skyline of the refined queries. Table 5.4 displays above the double line the skyline queries which also form the set of refined queries Q' finally returned to the user. Moreover, for each of the skyline queries, a score is assigned. The queries are displayed in order based on their score (from the best to the worst). We have assumed that each metric has the same weight.

For example, query Q'_{t_2} is dominated by Q'_{t_1} because Q'_{t_1} is better than Q'_{t_2} on the vd metric, and equal on all other metrics. Thus, Q'_{t_2} is not a skyline query and is not assigned with a score.

Let us calculate the score for one query, e.g., Q'_{t_1}. To compute the normalized metrics for Q'_{t_1}, we firstly find the maximum values among the skyline queries which are:

$$max_{ncc} = 2,\ max_{nac} = 2,\ max_{vd} = 3.6,\ max_{nfp} = 2$$

Thus, we have

$$score_{Q'_{t_1}} = (1 - 0.5) + (1 - 0) + (1 - 0,44) + (1 - 0.5) = 2.56$$

Finally, the precision of a query refinement is computed by the formula:

$$Presicion(Q) = \frac{\mid Q'[\mathcal{I}] \mid - FP'_Q}{\mid Q'[\mathcal{I}] \mid}$$

where FP'_Q is the number of false positive tuples output by Q'.

5.6 EFQ Platform

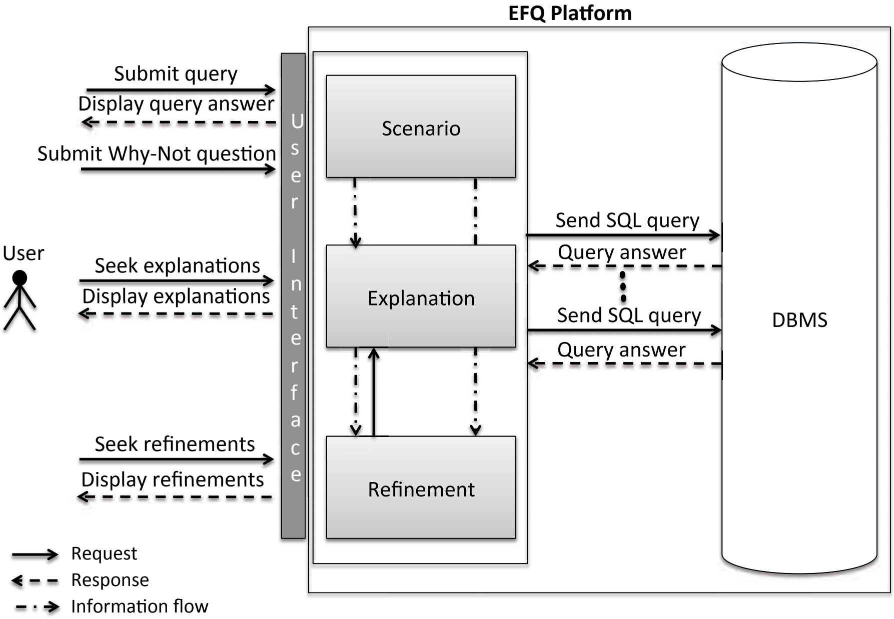
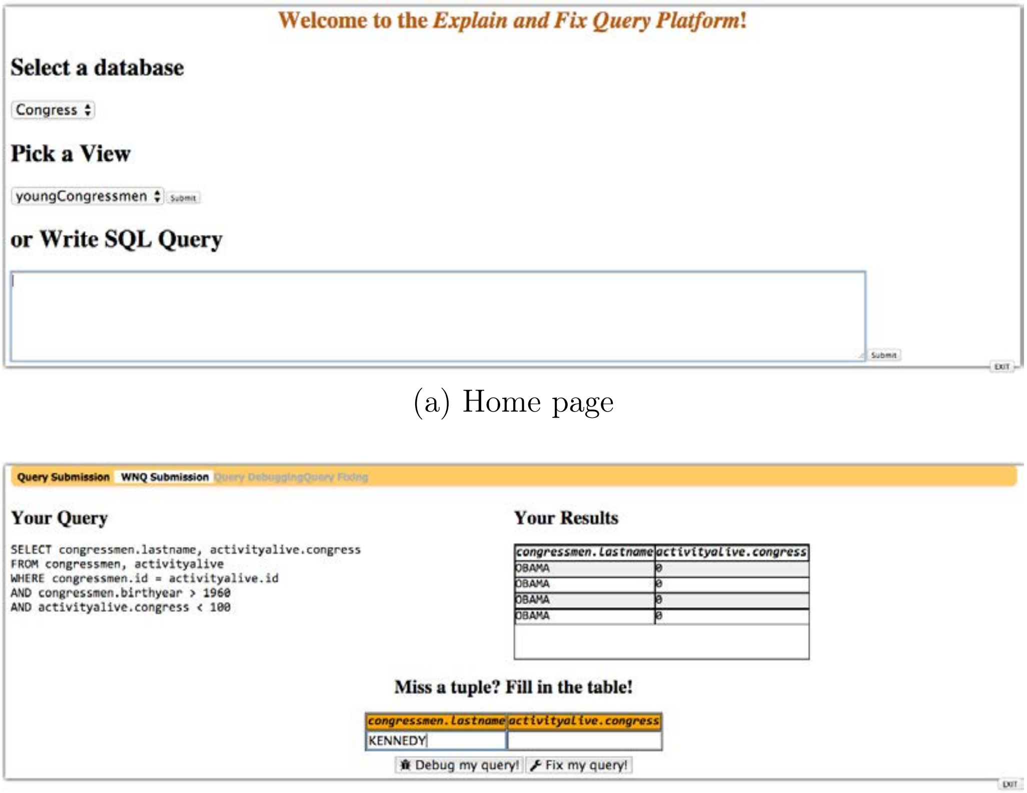


Figure 5.8: *EFQ* platform overview

In order to prove the feasibility of the proposed query debugging and fixing framework, we have developed a platform called *Explain and Fix Query (EFQ) Platform. EFQ* considers as input an explanation scenario which is addressed in two main components. The first one provides query debugging capabilities, incorporating the *Ted++* algorithm. The second one builds on the Why-Not answer polynomial provided by *Ted++* in order to suggest refined queries based on the *FixTed* algorithm.

5.6.1 Set up

All the involved algorithms in *EFQ* are implemented in Java. The underlying database system is PostgresSQL 9.3 (www.postgresql.org/). The system runs as a local apache (http://www.apache.org/) web application built using JavaServer Pages (JSP) Technology (http://www.oracle.com/technetwork/java/javaee/jsp/index.html).



Welcome to the *Explain and Fix Query Platform*!

Select a database

Congress

Pick a View

youngCongressmen

or Write SQL Query

(a) Home page

Your Query

```
SELECT congressmen.lastname, activityalive.congress
FROM congressmen, activityalive
WHERE congressmen.id = activityalive.id
AND congressmen.birthyear > 1960
AND activityalive.congress < 100
```

Your Results

congressmen.lastname	activityalive.congress
OBAMA	0
OBAMA	0
OBAMA	0
OBAMA	0

Miss a tuple? Fill in the table!

congressmen.lastname	activityalive.congress
KENNEDY	

(b) Scenario component

Figure 5.9: *EFQ* home and Scenario pages

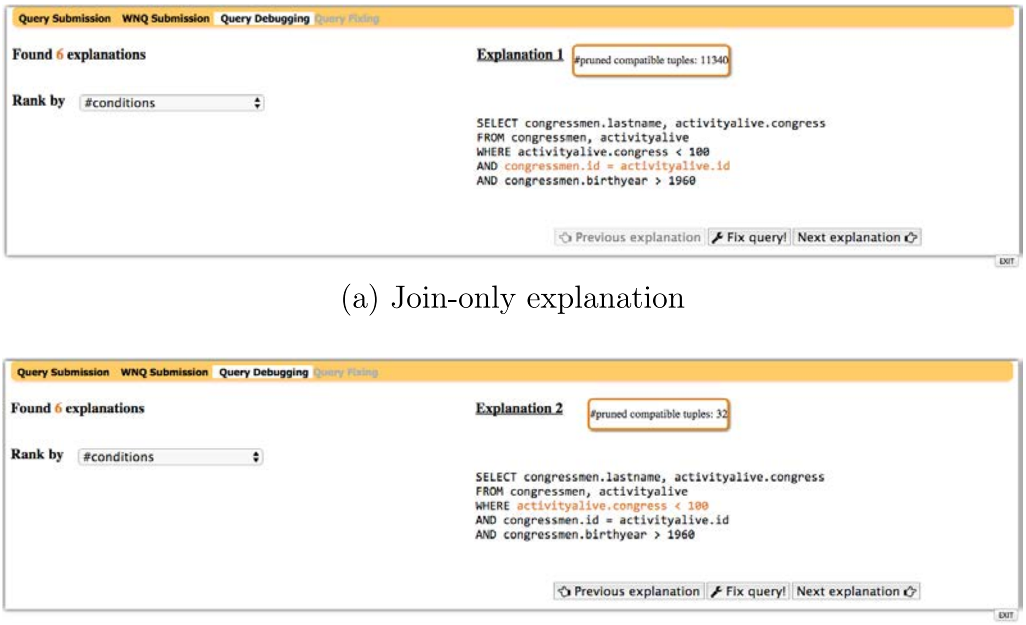
5.6.2 Platform description through a use case

The *EFQ* platform [BHT15c] facilitates the query debugging and fixing experience of a user through a series of interactions. The architecture of the system as well as the main actions and information flow between the user and the platform components is shown in Figure 5.8. There are three basic components in *EFQ*: (1) the Scenario component, (2) the Explanation component and, (3) the Refinement component. In the following, we discuss the three main components in more detail, guided through a use case.

Use case: Kennedy is missing!

Let us consider a user that is interested in young congressmen of the United States, who have participated in recent congresses. To make this query, the user can visit the home page of *EFQ* (Figure 5.9(a)), select the related database *Congress* and also the related database view *YoungCongressmen*. Otherwise, she can type the query in the dedicated box. By submitting the query, the user is directed in the *Scenario* component interface (Figure 5.9), where the query result is displayed. In this case, only *Obama* is returned as an answer. The user, knowing the family tradition of the Kennedies in political life, wonders why there is no *Kennedy* in the result. To express this Why-Not question, she fills in the table with the value *Kennedy*.

Then, since there are compatible tuples for the Why-Not question, the platform allows for two options: ask for (i) explanations, and/or (ii) query refinements.



Found 6 explanations

Rank by #conditions

Explanation 1 #pruned compatible tuples: 11340

```
SELECT congressmen.lastname, activityalive.congress
FROM congressmen, activityalive
WHERE activityalive.congress < 100
AND congressmen.id = activityalive.id
AND congressmen.birthyear > 1960
```

(a) Join-only explanation

Found 6 explanations

Rank by #conditions

Explanation 2 #pruned compatible tuples: 32

```
SELECT congressmen.lastname, activityalive.congress
FROM congressmen, activityalive
WHERE activityalive.congress < 100
AND congressmen.id = activityalive.id
AND congressmen.birthyear > 1960
```

(b) Selection-only explanation

Figure 5.10: *EFQ* Explanation component

Explanations

Figure 5.10 displays the interface for the *Explanation* component. Here, the platform displays the resulting Why-Not answer polynomial computed by *Ted++*, the terms of which correspond to the different explanations. The user can navigate in this page through the explanations, ordered by (1) the number of conditions in the explanation, or (2) the upper bound of the number of recoverable missing tuples. For each explanation, the involved conditions are highlighted on the query statement, while also the upper bound of the number of recoverable tuples is displayed. This number is computed based on the coefficients of the explanations, as discussed in Definition 4.2.3 in Section 4.2.3. For example, *EFQ* has computed six different explanations. Figure 5.10(a) displays one of the explanations with the least size (=1), highlighting the join, which is responsible for pruning out of the result 11340 compatible tuples. This is also the upper bound of the number of recoverable missing tuples, if this join is fixed. Another possible explanation including only one condition, is the selection $\sigma_{congress<100}[activityalive]$ (Figure 5.10(b)).

Refinements

Figure 5.11 demonstrates the interface of the *Refinement* component. The query refinements are computed based on the Why-Not answer polynomial by the *FixTed* algorithm. As such, the user can opt either for the full set of refined queries or filtered by explanation. As there may be numerous proposed refinements, the user can select the order in which they are presented, based on (1) their similarity with the original query, depending on the number of changed conditions, distance of the changed constant value from the original one, number of added conditions on attributes not constrained in the original query and type of involved conditions (joins or selections) or on (2) precision, measuring how many false positive tuples appear in the result of the new query.

Furthermore, the user can customise the underlying cost function by adjusting the weights of each parameter.

For the user's convenience, the interface highlights in each refinement the changes made in the conditions, so that the user quickly understands the alterations w.r.t. the original query.

For example, consider that the user wants a refinement based on the explanation in Figure 5.10(b), which is a selections-only explanation. Figure 5.11(a) shows the refinement in which the selection $activityalive.congress < 100$ has been changed to $activityalive.congress < 100 \leq 104$. The user observes that the precision is low, so she would like to see more precise refinements. *EFQ* proposes two more refined queries (see Figure 5.11(b)-(c)). As the metric values show, the false positive tuples are indeed fewer, which however takes its toll on the similarity of the queries.

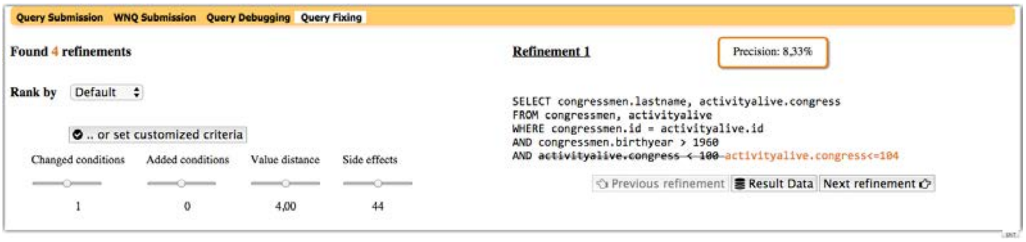
Now, let us consider that the user wants to find out how she could fix the (explanation including the) join (Figure 5.10(a)). Figure 5.11(d) shows the solution using a left outer join. It has been possible to provide such a solution, because the query and the explanation conform with the requirements described in Section 5.4.3. Note that the refined query involves also two sub-queries, one for each relation over which selection conditions are specified in the original query. On the same page, the user is also informed about the values of the metrics for the current refinement, as well as its precision. The precision for this query is low, because of the number of result tuples generated because of the left outer join. The similarity of the refined query appears to be low, but in fact only one condition is changed (the join). The value distance is zero, because no constants were altered in the refined query. Note here that for the joins-only refinements, the change is not highlighted on the interface, and that the visualization is to be improved.

Even though *EFQ* proposes a number of refined queries, in the end it is up to the user to pick the one that best meets her needs and expectations. Furthermore, the refinements, along with the explanations, can be used as a guidance for the fixing task of the user, if she wants to further or differently refine the query.

5.7 Summary and Future Work

In this chapter, we have proposed refinement-based explanations to solve the query fixing problem. Our approach takes into account previously generated query-based explanations, and thus is capable of providing targeted query refinements to specific query-based explanations and explanation types. Specifically for explanations involving joins, we proposed a novel way of repairing the query using left or right outer joins. For selections, we showed how the knowledge of the query-based explanation can lead to most similar query refinements. Moreover, we provided *FixTed*, an algorithm that computes such query refinements and we showed that *FixTed* has the potential to reduce the computational complexity w.r.t. the approach not considering query-based explanations. Finally, we demonstrated the *EFQ* platform integrating both query-based explanations and query refinements to provide a common tool for query debugging and query fixing in the context of SQL queries.

In the future several improvements can be made to *FixTed.* First of all, when explanations involving joins are considered we are envisaging to propose also (when possible by the database schema) alterations not only on the query condition set, but also on the *from* clause of the query. To do this, we can use integrity constraints, such as foreign keys. Moreover, even though a first validation of the *FixTed* algorithm has been performed through the *EFQ* application, we plan to do a more thorough experimental validation. This experimental validation could be coupled with a user study that would help evaluate the usefulness of the provided refined queries, by real users. Finally, another line of work would expand the class of considered queries in the input of the algorithms, so that outer joins are included. In this way, the user could continue the query analysis in the case when an outer-join refinement is proposed.



Found 4 refinements

Rank by Default

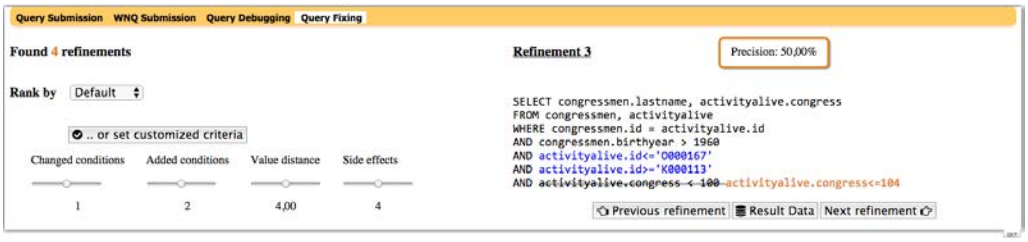
Changed conditions	Added conditions	Value distance	Side effects
1	0	4,00	44

Refinement 1

Precision: 8,33%

```
SELECT congressmen.lastname, activityalive.congress
FROM congressmen, activityalive
WHERE congressmen.id = activityalive.id
AND congressmen.birthyear > 1960
AND ~~activityalive.congress < 100~~ activityalive.congress<=104
```

(a) Selections-only refinement (highest similarity)



Found 4 refinements

Rank by Default

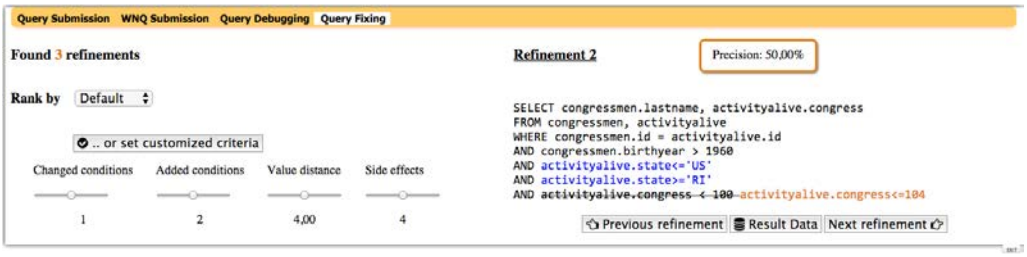
Changed conditions	Added conditions	Value distance	Side effects
1	2	4,00	4

Refinement 3

Precision: 50,00%

```
SELECT congressmen.lastname, activityalive.congress
FROM congressmen, activityalive
WHERE congressmen.id = activityalive.id
AND congressmen.birthyear > 1960
AND activityalive.id<='O000167'
AND activityalive.id>='K000113'
AND ~~activityalive.congress < 100~~ activityalive.congress<=104
```

(b) Selection-only refinement (improved precision)



Found 3 refinements

Rank by Default

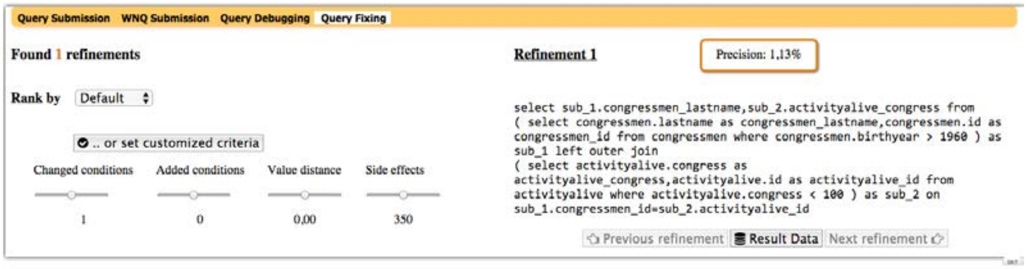
Changed conditions	Added conditions	Value distance	Side effects
1	2	4,00	4

Refinement 2

Precision: 50,00%

```
SELECT congressmen.lastname, activityalive.congress
FROM congressmen, activityalive
WHERE congressmen.id = activityalive.id
AND congressmen.birthyear > 1960
AND activityalive.state<='US'
AND activityalive.state>='RI'
AND ~~activityalive.congress < 100~~ activityalive.congress<=104
```

(c) Selections-only refinement (improved precision)



Found 1 refinements

Rank by Default

Changed conditions	Added conditions	Value distance	Side effects
1	0	0,00	350

Refinement 1

Precision: 1,13%

```
select sub_1.congressmen_lastname,sub_2.activityalive_congress from
( select congressmen.lastname as congressmen_lastname,congressmen.id as
congressmen_id from congressmen where congressmen.birthyear > 1960 ) as
sub_1 left outer join
( select activityalive.congress as
activityalive_congress,activityalive.id as activityalive_id from
activityalive where activityalive.congress < 100 ) as sub_2 on
sub_1.congressmen_id=sub_2.activityalive_id
```

(d) Joins-only refinement

Figure 5.11: *EFQ* Refinement component

Chapter 6

Conclusion and Future Work

The technological advance in computer science has played an important role on the rapid evolution of other scientific fields like economics, biology, astronomy e.t.c., as well as in business and trades markets. In any of these domains, scientists or businessmen are now in the position of producing, exchanging and combining more and more data not restricted to their fields. These data may be a result of 'manual' production, or may be automatically generated using for example programs, mash-up applications or workflows.

As a result, the origins of the contents of a database vary. There may be raw data, derived or extracted from various sources, or even resulting from some data transformation procedure over data that already may have been transformed. Consequently, end database users are in the need of mechanisms allowing them to verify the sanity of the database contents and to understand why and how these data were produced. After all, being able to trust the quality of the involved information is an important property that databases should provide users with.

To enhance databases with this property, researchers in the database domain have been working towards inventing the means and tools to trace back data to their origins and document the relevant information (which may vary from transformation steps to more general metadata). In one common term, information about the origins of data is defined as data provenance. Using data provenance the users can find out the reasons behind the existence of certain data in the database. Thus, in the case of expected data the users can reproduce them, or in the case of unexpected data they are able to correct the data or the data generation procedures. As a consequence, taking advantage of data provenance, tasks like data reproducibility, integration or curation and cleaning are rendered easier and more effective.

However, data missing from a database instance are as important as existing data. Thus, as was the case for existing data, our goal is to explain why expected data are not output by data transformations and consequently be able to fix this problem. This thesis focuses on devising methods and mechanisms to address this particular problem of relational queries (specifying for example data transformations) not returning expected results.

We typically specify missing results using Why-Not questions. The answer to a Why-Not question can be identified as the tuples that should be updated in the database in order to make it possible to recover missing tuples, which compose the so-called instance-based explanations. Alternatively, the answer to a Why-Not question may identify the problems in the query that lead to discarding data relevant to the missing tuples, which compose the so-called query-based explanations. While instance-based explanations debug the input database instance, query-based explanations are used to debug the query producing the result. This is important in the case when the input database is trusted and not subject to changes, and/or the involved queries are used for complex data analysis tasks.

In this thesis, we considered a user 'trapped' in such a scenario of having a query that does not output some tuples that she expects. We firstly proposed two methods to semi-automatically debug the query by computing query-based explanations. Guided by the query-based explanations, the user may subsequently opt to manually repair the query. However, a query refinement process, i.e., finding how to repair a query, can be a as or even more task than a query debugging process. For this reason, secondly we proposed a method to semi-automatically repair the query in order to recover missing tuples, based on query-based explanations. Based on query pruning techniques, we only return the most useful query refinements, based on criteria like syntax similarity and result precision. The user is then able to choose from the returned query refinements the one that best fits her expectations.

Next, we summarize this thesis content describing our contribution to the query debugging and fixing problem to recover missing results. Subsequently, we discuss interesting perspectives and open questions for future investigation.

6.1 Thesis Summary

In this thesis, we have presented our contribution in two phases, concerning the specific problem we addressed.

Query Debugging Our first contribution consists in providing the formal framework around Why-Not questions and relevant notions, missing from the related work. Then, we propose two approaches to answer Why-Not questions with query-based explanations, generally referred to as *NedExplain* and *Ted*. More specifically, each proposal makes the following contributions:

1. **Tree-based approach** We proposed an approach for computing query-based explanations that is inspired by [CJ09]. As such, it is based on a specific tree representation of the query. Firstly, we reviewed in detail the proposed solution in [CJ09]. We showed in theory and in practice that [CJ09] demonstrates a number of shortcomings related to how the source data relevant to the Why-Not question are computed and traced on the query tree. More specifically, we showed how the definition proposed by [CJ09] in certain cases leads to inaccurate or incomplete query-based explanations, or even worse in some cases are inadequate to identify any query-based explanation. Then, we proposed a new formal framework to capture all the cases missed by [CJ09] and moreover to accommodate for aggregate queries in addition to the class of select-project-join-union queries handled by [CJ09]. Furthermore, we devised and implemented the *NedExplain* algorithm to compute query-based explanations based on our formal framework. Finally, we demonstrated by a comparative evaluation that the query-based explanations returned by *NedExplain* are superior to those returned by [CJ09] both in terms of quality and run time.

   (a) *Formalization of query-based why-not provenance.* We provided a formalization of query-based explanations for Why-Not questions that was missing in [CJ09]. It relies on new notions of compatible tuples and of their valid successors. This definition subsumes the concepts informally introduced previously. It covers cases that were not properly captured in [CJ09]. Moreover it takes into account queries involving aggregation (i.e., select-project-join-aggregate queries, or SPJA queries for short) and unions thereof.

   (b) *The* NedExplain *Algorithm.* Based on the problem formalization, the *NedExplain* algorithm is designed to correctly compute query-based explanations given an explanation scenario over the class of unions of SPJA queries and a Why-Not question as in Definition 2.2.2.

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

   (d) *Detailed analysis of Why-Not.* We reviewed in detail Why-Not [CJ09] in the context of positive relational queries and showed that it has several shortcomings leading it to return no, partial, or misleading explanations.

2. **Polynomial-based approach** We introduced a new approach to computing query-based explanations as polynomials, relying on queries rather than query trees. As we show, the tree-based approach leads to incomplete results as far as the explanations are concerned. First, an explanation should indicate the complete set of conditions in it, and not sub-parts. Second, the complete set of explanations should be returned, and not only one explanation. These shortcomings of *NedExplain* (and in general of the query-tree approach) are dealt with in *Ted.* Furthermore, we discussed the formalization of the Why-Not answer polynomials in different aspects. Two algorithms were proposed, the naive and straightforward *Ted*, which is shown to be inefficient in practice, and the optimized *Ted++* that renders the solution practically interesting too. Our experiments show that *Ted++* outperforms the related algorithms both in answer quality and in run times. In more detail, our contributions with the polynomial-based approach are:

   (a) *Why-Not answer polynomial.* Our formal framework is defined for conjunctive queries with inequalities and unions thereof for the relational data model under set semantics. It supports a larger class of Why-Not questions w.r.t. previous works, i.e., both simple and complex Why-Not questions. The form of the Why-Not answer is unprecedented, as this work is the first to formalize Why-Not answer polynomials providing fine-grained query based explanations. Intuitively, each term of a polynomial represents one combination of the query conditions that *together* explain the absence of some of the missing tuples and the set of all terms covers all possible such combinations. The coefficients can be used to obtain an upper bound on the number of recoverable missing tuples when properly changing the conditions of a term.

   (b) *Extended formalization of the Why-Not answer polynomial.* An extended formalization of Why-Not answer polynomials is provided to cover the relational data model under bag and probabilistic semantics. This confirms the robustness of the chosen polynomial representation, making it a good fit for a unified framework for representing query based explanations for different semantics.

(c) *Equivalent queries w.r.t. the Why-Not answer polynomial* We define the class of equivalent queries w.r.t. the Why-Not answer polynomial. We show that isomorphic queries have isomorphic Why-Not answer polynomials given a fixed Why-Not question. This leads us to state that the Why-Not answer polynomial is invariant of the topology of the query tree and moreover that the Why-Not answer polynomial subsumes the query-based explanations computed by *NedExplain.* Additionally, we show that equivalent queries in general are not equivalent w.r.t. the Why-Not answer polynomial and as such, no homomorphisms exist mapping the Why-Not answer polynomials of two equivalent queries to one another. Finally, we provide an approximate Why-Not answer polynomial for a query Q, when one equivalent query Q' resulting from tableau minimization of Q is available.

(d) *Naive* Ted *and optimized* Ted++ *algorithms.* We first provide a naive algorithm for computing Why-Not answer polynomials, named *Ted.* This algorithm is a straightforward implementation of the formal definitions. We show that *Ted* is impractical. We thus propose an optimized algorithm, *Ted++*, capable of efficiently computing the Why-Not answer polynomial, relying on schema and data partitioning (allowing for a distributed computation) and advantageous replacement of expensive database evaluations by mathematical calculations.

(e) *Experimental validation.* We experimentally evaluate the quality of the proposed Why-Not answer polynomial and the efficiency of the *Ted++* algorithm. The experiments include a comparative evaluation to existing algorithms computing query-based explanations for SQL queries in terms of explanation quality and run-time, as well as a thorough study of the *Ted++* performance w.r.t. different parameters.

Query Fixing We proposed a new approach towards fixing conjunctive queries with inequalities by building query refinements. This approach exploits the query-based explanations modelled as terms in a Why-Not answer polynomial. We proposed two main types of query refinements, the one associated with explanations containing selections and the other associated with explanations containing joins. For explanations with joins, we proposed a novel type of query refinements using outer joins. We further proposed the techniques to compute query refinements, embodied in the *FixTed* Algorithm and we discussed how our techniques of computing query refinements has the potential to be more efficient than the related work. Finally, we integrated the implementation of *FixTed* in our *EFQ* Platform and showed that the Why-Not answer polynomial can be used for query debugging and fixing in practice in an interactive environment. In more detail, our contributions in the part of Query Fixing are the following:

1. *Query refinements based on query-based explantions.* We provide the framework to compute query refinements for conjunctive queries with inequalities. We are the first to leverage query-based explanations in the form of Why-Not answer polynomials

to compute the query refinements. In this way we succeed to compute the most similar query refinements possible. Moreover, the polynomial provides us with the information which conditions to focus on, which allows us to reduce the search space for query refinements, ultimately leading to a more efficient solution (as opposed to an approach not considering query-based explanations).

2. *Query refinements and explanation type* A query-refinement obtained by our method is linked to an explanation (term) of the Why-Not answer polynomial. We distinguish between two main categories of query refinements, depending on the type of the explanation.

    (a) *Explanations with selection conditions* The first one considers explanations containing selection conditions and in principal consists in conjunctive queries having the explanation selection conditions fixed. To obtain these, we proceed in two steps, exploiting the technique of skyline tuples. More specifically, we propose a variation of skyline that potentially leads to tackling the selections refinement problem more efficiently. The first step is designed so as to perform the least changes to the selections in the explanations needed to allow missing tuples appear in the result. This step lead to obtaining the most similar refinements to the query (per explanation). However, by relaxing the conditions of the query, we have as a consequence that not only missing tuples but also irrelevant tuples are added to the refined query result. As a remedy, we proceed to the second step that eliminates irrelevant (a.k.a. false positive) tuples from the refined result. This step consists in further refining the query by adding progressively conditions to the refinements produced in the first step.

    (b) *Explanations with join conditions* The second category of query refinements considers explanations with join conditions. In this case, we introduce left or right outer joins in the query refinement and thus move to a wider class than conjunctive queries. To the best of our knowledge we are the first to propose such kind of refinements. We show however that we cannot refine all conjunctive queries using this approach. Indeed, in the graph modelling the joins in the query, two direct relations (a.k.a., relations over which the Why-Not question is defined) should not be connected via a 'picky' join. For this reason, we investigate when such refinements are feasible for a query and propose a way to refine queries with outer joins, given the query and the explanation.

3. *FixTed Algorithm and EFQ Platform* We provide an algorithm, *FixTed* to compute query refinements w.r.t a query and a Why-Not answer polynomial, as defined by our framework. We discuss how *FixTed* also defines a ranking function to order the best refinements, which are in the skyline of queries w.r.t. similarity and precision. Then, we demonstrate the practicality of our proposal by incorporating the *Ted++* algorithm along with the *FixTed* algorithm into a novel platform named *EFQ*. *EFQ* provides the means for semi-automatically debugging and refining SQL queries, through an interactive interface.

6.2 Perspectives

We outline here some perspectives to this thesis work regarding the different thematics addressed.

Why-Not questions In this thesis, we have assumed that the user knows the domain of the database and that she has an idea of what results her queries should return. In this way, we consider that the user can judge which results are considered correct or not. More specifically in our context the user is able to define which results are missing and thus to state Why-Not questions about these. Even though this assumption might be acceptable in cases when the users are experts or the involved data few enough to be managed by a human, asking Why-Not questions might itself constitute a problem of research. Thus, one possible line of work would investigate how to indicate to the users the results that might be missing. This could be done in an interactive environment where the system guides the user to locate the data properties that could be interesting to her. Such suggestions include schema attributes, sample values or conditions over the attributes, built progressively by combining the user's response with the database information. Moreover, the system could function like a recommendation system [RRS11], taking into account also her previous Why-Not questions or other users' Why-Not questions as well.

Query debugging In Chapter 4 we proposed two alternative ways of computing query-based explanations and we outlined the features of each one. To elaborate even more the difference of the proposals w.r.t. the answer quality and algorithms' efficiency, an extension to the experimental part could be added. An interesting experiment would be to run *NedExplain* with all different trees with reordered operators and compare the obtained answers with the one returned by *Ted*. The expected result of the experiment is that still *NedExplain* fails to compute all the explanations indicated by *Ted*.

As far as the tree-based approach is concerned, we could extend *NedExplain* to generate 'ghost' tuples for the tuples lost in the nodes identified as picky, a technique already used in [Her15]. As eventually all the nodes will be checked, this addition could cope with an aspect of the answer completeness problem.

Further interesting extensions include adding set difference to the query or adapting the method for complex Why-Not questions. Set difference would require tracing data that needs to make it to the result (compatible tuples) but also data that should not make it (in order not to eliminate the compatible tuples). So, Why-Not questions over queries with set difference divide the problem to both Why (or How) and Why-Not question provenance problems. However, Why (or How) provenance is an instance-based method and so a hybrid method seems more appropriate for queries with negation.

To adapt *NedExplain* for complex Why-Not questions we could re-design the tree such that it assigns partitions instead of relations to the leaves. Then, joins among relations of the same partition could be modelled as selection operators with complex conditions. Given this tree, *NedExplain* algorithm could be applied as usual.

As far as the Why-Not answer polynomial approach is concerned, the most interesting extension concerns the class of queries which is now limited to SPJ queries. In many applications, aggregate, nested or negative queries are most frequent and also more difficult to debug. For aggregate queries instead of considering individually the compatible tuples, an approach based on appropriate sets of compatible tuples satisfying the Why-Not question could be considered.

For nested queries a simple approach would be to un-nest the query and then apply the algorithm *Ted++*. However, it is not always possible to un-nest a query, so such a solution would not be general. Moreover an explanation on the un-nested query would have only little connection with the input query and thus it may be too difficult to understand by a user. Thus, it would be interesting to investigate a more general and useful approach.

In general, it is well known that negation makes queries difficult to process and to reason about [CH80, Bid91, BTCO12]. More particularly, the same holds for negation in queries and data provenance. The negative results on expressing provenance semirings with negative queries, provided by Amsterdamer et.all [ADT11a], show that even though provenance semirings are a nice common framework for different semantics in the case of positive queries, this solution cannot be generalized for negative queries. Based on this result, one expects that the extended Why-Not answer polynomial that we introduced for set, bag and probabilistic semantics would not be feasible for negative queries. Thus, it would be interesting to investigate if this intuition holds, and despite the possibly negative result, if Why-Not answer polynomials can be considered with negative queries under only set semantics.

It is worth noticing however, that there is a case when we can trivially extend Why-Not answer polynomial (and *NedExplain*) for queries with negation and set semantics. As an example, consider the following SQL query

```
SELECT Book.Title
FROM Book
WHERE Book.Author NOT IN {‘Homer’,‘Sophocles’}
```

In this case the negation `Book.Author NOT IN {‘Homer’,‘Sophocles’}` is equivalent to `Book.Author !=‘Homer’ AND Book.Author !=‘Sophocles’`. The latter, is a case treated by our algorithms.

Query fixing In Chapter 5 we proposed an approach to refining SPJ queries using explanations, and we provided the *FixTed* algorithm. Even though we demonstrated the feasibility of *FixTed* in our *EFQ* platform, still an experimental evaluation, comparative to related work algorithms like [TC10] is pending. This experimental evaluation should investigate the run time efficiency of *FixTed* and as expected show that our local skylines technique boosts the efficiency. For the usability and relevance of the proposed refinements, a user study would be the most appropriate evaluation method.

Moreover, we plan to improve/enhance our refining methods. As we described, refining an explanation with joins using outer joins may not always be applicable. Even more, such a solution is acceptable only when the user is interested in translating her query in the way that outer joins do. Thus, we plan to enrich the proposed refinements with more ways of repairing joins. A possible way to go is by using foreign key dependencies that will yield query refinements with possibly a different set of input relations (FROM clause). Another possibility is to check in for matching values in the active domain, in a way of discovering inclusion dependencies that could be used in the absence of foreign keys. Alternatively, following the approach of refining selections with $=$ using $\leq$ or $\geq$, we could transform equality joins to theta joins. It would be interesting to experiment with various refining alternatives and see which ones are most efficient and which are most preferred by the end users, depending on different application domains.

Finally, the ranking function used to order the query refinements could be enriched with more parameters, like the type of the explanation, or the preference over the constrained attributes etc. Moreover, the similarity metric could be reconsidered after a user study evaluation, especially concerning the value distance metric.

Optimization techniques A main concern related to the algorithms proposed in this thesis is their scalability in the worst case. To deal with this problem we could investigate how moving to a column-store database would be more suitable for the algorithms, since the algorithms focus on certain attributes. Additionally, our algorithms allow for parallelizing the execution of specific tasks (like computing the eliminated compatile tuples per explanation for *Ted++* or the query refinements for each explanation in *FixTed*) a feature that has not been taken into consideration yet. A first attempt to using threads was done on a laptop with restricted processing capabilities, however using a cluster with several machines is expected to provide better results concerning the execution times.

Otherwise, when missing some query-based explanations is not an issue, we could devise heuristics taking into consideration only a sample of compatible data for the computation of the Why-Not answer polynomial. For *FixTed* we could prune the space of considered explanations by eliminating those that contain other explanations from the Why-Not answer polynomial, or considering only the explanations with the smallest size. Moreover, the user could specify a set of attributes which she would afford constraining in the refined queries, so as to reduce the complexity of *FixTed* and the number of returned refinements that would not be interesting for the user.

Why-Not questions in a different context In this work we have considered Why-Not questions in relational databases. For different data models like RDF [PFFC09, DAA12, GKCF13], data provenance has already been considered. Moreover, data provenance has been studied extensively for workflows [OAF$^+$04, LAB$^+$06, ICF$^+$12, CVKL$^+$14] Thus, it would also be relevant and interesting to study Why-Not provenance in a different context than relational databases and see if and how our algorithms can be adapted.

Bibliography

[ADD$^+$11] Yael Amsterdamer, Susan B. Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *PVLDB*, 5(4):346–357, 2011.

[ADT11a] Yael Amsterdamer, Daniel Deutch, and Val Tannen. On the limitations of provenance for queries with difference. In *TaPP*, 2011.

[ADT11b] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for aggregate queries. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 153–164. ACM, 2011.

[ASU79] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalences among relational expressions. *SIAM Journal on Computing*, 8(2):218–246, 1979.

[BFS00] Alain Bidault, Christine Froidevaux, and Brigitte Safar. Repairing queries in a mediator approach. In *ECAI*, pages 406–410, 2000.

[BHT13] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. Answering Why-Not questions. In *Base de données avancées (BDA)*, 2013.

[BHT14a] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. Immutably answering why-not questions for equivalent conjunctive queries. In *Workshop on Theory and Practice of Provenance (TAPP)*, 2014.

[BHT14b] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. Immutably answering why-not questions for equivalent conjunctive queries. In *Base de données avancées (BDA)*, 2014.

[BHT14c] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. Query-based why-not provenance with Nedexplain. In *International Conference on Extending Database Technology (EDBT)*, 2014.

[BHT15a] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. Efficient computation of polynomial explanations of why-not questions. In *International Conference on Information and Knowledge Management (CIKM)*, 2015.

[BHT15b] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. Efficient computation of polynomial explanations of why-not questions. In *Base de données avancées (BDA)*, 2015.

[BHT15c] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. Efq: Why-not answer polynomials in action. *Proceedings of the VLDB Endowment*, 8(12):1980–1983, 2015.

[BHT15d] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. Immutably answering why-not questions for equivalent conjunctive queries. *Ingénierie des Systèmes d'Information (ISI)*, 2015.

[Bid91] Nicole Bidoit. Negation in rule-based database languages: a survey. *Theoretical computer science*, 78(1):3–83, 1991.

[BKL07] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse query processing. In *ICDE*, pages 506–515, 2007.

[BKS01] S Borzsony, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 421–430. IEEE, 2001.

[BKT01] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[BMNT15] Moria Bergman, Tova Milo, Slava Novgorodov, and Wang-Chiew Tan. Query-oriented data cleaning with oracles. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1199–1214. ACM, 2015.

[BT07] Peter Buneman and Wang-Chiew Tan. Provenance in databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1171–1173. ACM, 2007.

[BTCO12] Vince Bárány, Balder Ten Cate, and Martin Otto. Queries with guarded negation. *Proceedings of the VLDB Endowment*, 5(11):1328–1339, 2012.

[CA15] Sean Chester and Ira Assent. Explanations for skyline query results. In *Extending Database Technology (EDBT)*, 2015.

[CB74] Donald D Chamberlin and Raymond F Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

[CCST15] Balder ten Cate, Cristina Civili, Evgeny Sherkhonov, and Wang-Chiew Tan. High-level why-not explanations using ontologies. In *Principles of Database Systems (PODS)*, pages 31–43, 2015.

[CCT09] James Cheney, Laura Chiticariu, and Wang Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4), 2009.

[CH80] Ashok K Chandra and David Harel. Structure and complexity of relational queries. In *Foundations of Computer Science, 1980., 21st Annual Symposium on*, pages 333–347. IEEE, 1980.

[CJ09] Adriane Chapman and H. V. Jagadish. Why not? In *International Conference on the Management of Data (SIGMOD)*, 2009.

[CLJX15] Lei Chen, Xin Lin, Christian S. Jensen, and Jianliang Xu. Answering why-not questions on spatial keyword top-k queries. In *International Conference on Data Engineering (ICDE)*, pages 279–290, 2015.

[CM77] Ashok K Chandra and Philip M Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90. ACM, 1977.

[Cod70] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod72] Edgar F Codd. *Relational completeness of data base sublanguages.* IBM Corporation, 1972.

[COSS13] Diego Calvanese, Magdalena Ortiz, Mantas Simkus, and Giorgio Stefanoni. Reasoning about explanations for negative query answers in dl-lite. *Journal on Artificial Intelligence Research (JAIR)*, 48:635–669, 2013.

[CR97] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. In *Database Theory—ICDT'97*, pages 56–70. Springer, 1997.

[CVKL$^+$14] Víctor Cuevas-Vicenttín, Parisa Kianmajd, Bertram Ludäscher, Paolo Missier, Fernando Chirigati, Yaxing Wei, David Koop, and Saumen Dey. The pbase scientific workflow provenance repository. *International Journal of Digital Curation*, 9(2):28–38, 2014.

[CW00] Yingwei Cui and Jennifer Widom. Practical lineage tracing in data warehouses. In *International Conference on Data Engineering (ICDE)*, 2000.

[CW01] Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. In *VLDB Conference*, 2001.

[CWW00] Yingwei Cui, Jennifer Widom, and Janet L Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179–227, 2000.

[DAA12] Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. Provenance for sparql queries. In *The Semantic Web–ISWC 2012*, pages 625–640. Springer, 2012.

[DAA13] Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. Justifications for logic programming. In *Logic Programming and Nonmonotonic Reasoning*, pages 530–542. Springer, 2013.

[DG11] Jonathan Danaparamita and Wolfgang Gatterbauer. QueryViz: helping users understand SQL queries and their patterns. In *International Conference on Extending Database Technology (EDBT)*, 2011.

[DS07] Evangelos Dellis and Bernhard Seeger. Efficient computation of reverse skyline queries. In *Proceedings of the 33rd international conference on Very large data bases*, pages 291–302. VLDB Endowment, 2007.

[FGT08] J Nathan Foster, Todd J Green, and Val Tannen. Annotated xml: queries and provenance. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 271–280, 2008.

[FKMP05] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1), 2005.

[GKCF13] Floris Geerts, Grigoris Karvounarakis, Vassilis Christophides, and Irini Fundulaki. Algebraic structures for capturing the provenance of sparql queries. In *Proceedings of the 16th International Conference on Database Theory*, pages 153–164. ACM, 2013.

[GKRS11] Torsten Grust, Fabian Kliebhan, Jan Rittinger, and Tom Schreiber. True language-level sql debugging. In *EDBT*, pages 562–565, 2011.

[GKT07] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *Principles of Database Systems (PODS)*, 2007.

[GLC$^+$15] Yunjun Gao, Qing Liu, Gang Chen, Baihua Zheng, and Linlin Zhou. Answering why-not questions on reverse top-k queries. *Proceedings of the VLDB Endowment*, 8(7):738–749, 2015.

[GM15] Benoit Groz and Tova Milo. Skyline queries with noisy comparisons. In *Proceedings of the 34th ACM Symposium on Principles of Database Systems*, pages 185–198. ACM, 2015.

[God97] Parke Godfrey. Minimization in cooperative response to failing database queries. *International Journal of Cooperative Information Systems*, 6(02):95–149, 1997.

[Hal98] Marshall Hall. *Combinatorial theory*, volume 71. John Wiley & Sons, 1998.

[HCDN08] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. On the provenance of non-answers to queries over extracted data. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1), 2008.

[HE12] Melanie Herschel and Hanno Eichelberger. The Nautilus Analyzer: understanding and debugging data transformations. In *International Conference on Information and Knowledge Management (CIKM)*, 2012.

[Her13] Melanie Herschel. Wondering why data are missing from query results? ask conseil why-not. In *International Conference on Information and Knowledge Management (CIKM)*, 2013.

[Her15] Melanie Herschel. A hybrid approach to answering why-not questions on relational query results. *ACM-JDIQ*, 5(3):10:1–10:29, 2015.

[HH10] Melanie Herschel and Mauricio A. Hernández. Explaining missing answers to SPJUA queries. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1), 2010.

[HL12] Zhian He and Eric Lo. Answering why-not questions on top-k queries. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, pages 750–761. IEEE Computer Society, 2012.

[HL14] Zhian He and Eric Lo. Answering why-not questions on top-k queries. *Knowledge and Data Engineering, IEEE Transactions on*, 26(6):1300–1315, 2014.

[ICF$^+$12] Robert Ikeda, Junsang Cho, Charlie Fang, Semih Salihoglu, Satoshi Torikai, and Jennifer Widom. Provenance-based debugging and drill-down in data-oriented workflows. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1249–1252. IEEE, 2012.

[ILZ12] Md Saiful Islam, Chengfei Liu, and Rui Zhou. On modeling query refinement by capturing user intent through feedback. In *Proceedings of the Twenty-Third Australasian Database Conference-Volume 124*, pages 11–20. Australian Computer Society, Inc., 2012.

[ILZ14] Md Saiful Islam, Chengfei Liu, and Rui Zhou. Flexiq: A flexible interactive querying framework by exploiting the skyline operator. *Journal of Systems and Software*, 97:97–117, 2014.

[IWL84] Tomasz Imieliński and Jr. Witold Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4), 1984.

[IZL13] Md. Saiful Islam, Rui Zhou, and Chengfei Liu. On answering why-not questions in reverse skyline queries. In *International Conference on Data Engineering (ICDE)*, 2013.

[KKBS10] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. SnipSuggest: Context-aware autocompletion for SQL. *Proceedings of the VLDB Endowment (PVLDB)*, 4(1), 2010.

[KLP75] Hsiang-Tsung Kung, Fabrizio Luccio, and Franco P Preparata. On finding the maxima of a set of vectors. *Journal of the ACM (JACM)*, 22(4):469–476, 1975.

[KLS12] Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis. *Declarative datalog debugging for mere mortals.* Springer, 2012.

[KLZ13] Sven Köhler, Bertram Ludäscher, and Daniel Zinn. First-order provenance games. In *In Search of Elegance in the Theory and Practice of Computation*, pages 382–399. Springer, 2013.

[LAB+06] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew B Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[MGMS11] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. The complexity of causality and responsibility for query answers and non-answers. *PVLDB*, 2011.

[MK09] Chaitanya Mishra and Nick Koudas. Interactive query refinement. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 862–873. ACM, 2009.

[MMR+13] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. A probabilistic optimization framework for the empty-answer problem. *Proceedings of the VLDB Endowment*, 6(14), 2013.

[NJ11] Arnab Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *Proceedings of the VLDB (PVLDB)*, 4(12), 2011.

[OAF+04] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[OCS09] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. Generating example data for dataflow programs. In *SIGMOD Conference*, pages 245–256, 2009.

[ÖÖM87] G Özsoyoğlu, ZM Özsoyoğlu, and Victor Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems (TODS)*, 12(4):566–592, 1987.

[PFFC09] Panagiotis Pediaditis, Giorgos Flouris, Irini Fundulaki, and Vassilis Christophides. On explicit provenance management in rdf/s graphs. In *Workshop on the Theory and Practice of Provenance*, 2009.

[PIW11] Hyunjung Park, Robert Ikeda, and Jennifer Widom. Ramp: A system for capturing and tracing provenance in mapreduce workflows. *PVLDB*, 4(12), 2011.

[PTFS03] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An optimal and progressive algorithm for skyline queries. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 467–478. ACM, 2003.

[Qui87] J. Ross Quinlan. Simplifying decision trees. *International journal of man-machine studies*, 27(3):221–234, 1987.

[RKL14] Sean Riddle, Sven Köhler, and Bertram Ludäscher. Towards constraint provenance games. In *Workshop on Theory and Practice of Provenance (TAPP)*, 2014.

[RRS11] Francesco Ricci, Lior Rokach, and Bracha Shapira. *Introduction to recommender systems handbook*. Springer, 2011.

[RS14] Sudeepa Roy and Dan Suciu. A formal approach to finding explanations for database queries. In *SIGMOD Conference*, 2014.

[ST12] Cheng Sheng and Yufei Tao. Worst-case i/o-efficient skyline algorithms. *ACM Transactions on Database Systems (TODS)*, 37(4):26, 2012.

[TC10] Quoc Trung Tran and Chee-Yong Chan. How to ConQueR why-not questions. In *International Conference on the Management of Data (SIGMOD)*, 2010.

[TCP09] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 535–548. ACM, 2009.

[TCP14] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query reverse engineering. *The VLDB Journal*, 23(5):721–746, 2014.

[Tzo14] Katerina Tzompanaki. Semi-automatic sql debugging and fixing to solve the missing-answers problem. In *Very Large Databases (VLDB'14) PhD Workshop*, 2014.

[Vau01] Robert Vaught. *Set Theory An Introduction*. Birkhaeuser, 2001.

[VDKN10] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Nørvåg. Reverse top-k queries. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 365–376. IEEE, 2010.

[WS97] Allison Woodruff and Michael Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, pages 91–102, 1997.

[ZDTT10] Feng Zhao, Gautam Das, Kian-Lee Tan, and Anthony KH Tung. Call to order: a hierarchical browsing approach to eliciting users' preference. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 27–38. ACM, 2010.

[ZHJ$^+$13] Jianfeng Zhang, Weihong Han, Yan Jia, Peng Zou, and Hua Fan. A novel approach to query modification based on user's why-not question. *International Journal of Computer Science Issues (IJCSI)*, 10(1), 2013.