Refining SQL Queries based on Why-Not Polynomials

Nicole Bidoit
LRI - Université Paris Sud,
CNRS (UMR8623)
Université Paris-Saclay
91405 Orsay Cedex, France
bidoit@lri.fr

Melanie Herschel
IPVS - University of Stuttgart
70569 Stuttgart, Germany
melanie.herschel@ipvs.uni-stuttgart.de

Katerina Tzompanaki *
INFRES-Télécom ParisTech
75013 Paris, France
aikaterini.tzompanaki@telecom-paristech.fr

Abstract

Explaining why some data are *not* part of a query result has recently gained significant interest. One use of *why-not* explanations is adapting queries to meet user expectations. We propose an algorithm to automatically generate changes to a query, by using *Why-Not polynomials*, one form of why-not explanations based on query operators. We improve on the state of the art in three aspects: (i) we refine both selection and join predicates, (ii) we guarantee a maximum similarity to the original query, and (iii) we cover all possible cases of why the desired data was missing. A prototype implementation shows the applicability of our approach in practice.

Categories and Subject Descriptors H.4 [*Information Systems Applications*]: Miscellaneous; D.2 [*Software Engineering*]: Testing and Debugging

Keywords data provenance, query analysis, why not questions

1. Introduction

Explaining why some data are *not* part of a query result has many applications ranging from information extraction (Huang et al. 2008) over query debugging (ten Cate et al. 2015) to commercial campaign and product tuning (Gao et al. 2015). For all these applications, understanding why expected answers are missing from a result typically precedes an adjustment process designed to meet the user expectation, based on the provided explanations. For instance, upon identifying which source data has not been properly extracted, the corresponding extractor is adapted.

This paper focuses on query repair, with the specific goal of semi-automatically adapting SQL queries to include expected (but missing) answers. Our approach builds on a previous contribution (Bidoit et al. 2014, 2015a) which, given a query Q and a Why-Not question provides a *Why-Not polynomial* that captures all possible explanations based on query operators. Essentially, the Why-Not polynomial pinpoints all combinations of query conditions (selections and joins) that potentially cause the missing answers. Based on these explanations, the *FixTed* algorithm discussed in this paper refines the original query Q to include the missing answers.

* Work started while at LRI - Université Paris Sud.

TaPP 2016, June 8–9, 2016, Washington, DC.

Movie

title	rating	length	date	
$Revenant$	8.2	156	2015	Id_1
$J.Edgar$	7.6	137	2011	Id_2
$S.Island$	8	138	2010	Id_3
$Titanic$	8.1	194	1997	Id_4
$Fury$	8.1	134	2014	Id_5
$Matrix$	8.7	136	1999	Id_6

Actor

name	age	film	
$DiCaprio$	42	$Revenant$	Id_7
$DiCaprio$	38	$J.Edgar$	Id_8
$DiCaprio$	36	$S.Island$	Id_9
$DiCaprio$	24	$Titanic$	Id_{10}
$Lerman$	22	$Fury$	Id_{11}
$Reeves$	34	$Matrix$	Id_{12}

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

```
SELECT A.name, M.title
FROM Movie M, Actor A
WHERE A.age ≤ 35 (c1)
   AND M.rating > 8.2 (c2)
   AND A.film = M.title (c3)
```

(b) Sample query Q

```
SELECT A.name, M.title
FROM Movie M, Actor A
WHERE A.age ≤ 35
   AND M.rating ≥ 8.1
   AND A.film = M.title
```

(c) Refinement of Q w.r.t. c_2

$$c_2 + c_3 + 3c_1c_2 + 4c_2c_3 + 3c_1c_3 + 12c_1c_2c_3$$

(d) Why-Not polynomial: each addend provides an explanation

Figure 1. Running example

Example 1.1. *The SQL query of Fig. 1(b) returns actors who starred in excellent movies while still young. Query conditions are denoted c_1, c_2, and c_3. Given the query result $\{(Reeves, Matrix)\}$, one may ask the Why-Not question "Why not DiCaprio?". The first reason that comes to mind is that DiCaprio is not that young, i.e., c_1 may be too selective. But let us now analyze the explanations provided by the Why-Not polynomial of Fig. 1(d), where each addend provides one explanation (a product of query conditions) for missing DiCaprio. We observe that c_1 appears in three explanations, but always in combination with another operator, e.g., in the explanation c_1c_2. This entails that the operator c_1 is never responsible alone for pruning DiCaprio from the result. So, if we opt for changing c_1, we would also need to change also c_2 to obtain DiCaprio in the result. The addend $3c_1c_2$ tells us also that there exist three different ways to adjust the values of c_1c_2, corresponding to the tuple pairs Id_1Id_7, Id_2Id_8, and Id_3Id_9. On the other hand, the addend c_2 of the Why-Not polynomial, indicates that there is one possibility to recover DiCaprio by just relaxing the condition on rating. The corresponding refinement is shown in Fig. 1(c).*

Contributions. Existing work on query refinement addressing Why-Not questions (Gaasterland et al. 1992; Islam et al. 2013; He and Lo 2014; Chen et al. 2015; Gao et al. 2015), especially for relational queries (Tran and Chan 2010; Tran et al. 2014; Islam et al. 2012, 2014), directly compute query refinements ignoring completely which operators or combination of operators explain the missing data. However, such information, which is provided by Why-Not polynomials, is valuable for guiding the refinement process of the original query. Indeed, explanations pinpoint the query conditions that need to be changed, reducing the search space explored by other approaches. *FixTed* is the first algorithm that **leverages query-based explanations to compute query refinements**.

Query	Query result
User query $Q{=}(\mathcal{S}, \Gamma, C)$	Answer tuples : $Q[\mathcal{I}]$
Why-Not question $WN{=}(\mathcal{S}, \Gamma, C_W)$	Missing tuples : $WN[\mathcal{I}]$
Extended Why-Not question $eWN(\mathcal{S}, Attr(\mathcal{S}), C_W)$	Compatible tuples (CT): $eWN[\mathcal{I}]$

Table 1. Preliminary queries and tuples

In addition, *FixTed* is the first query refinement algorithm for relational queries to **rewrite joins into outer joins**, thereby covering a typical source of error and enlarging the set of possible query refinements over the state of the art. Finally, the set of generated refinements guarantees two important properties not enforced by prior work. First, *FixTed* creates **refinements with maximum similarity** to the original query. Second, *FixTed* ensures that **all possible ways to repair the original query** are investigated and thus returns a comprehensive set of refinements.

2. Preliminaries and Problem Statement

FixTed relies on various concepts formalized in prior work (Bidoit et al. 2015a). Due to space constraints, we limit the discussion here to an overview of the most central concepts. Tab. 1 summarizes the different queries and query results we rely on, as discussed below.

We are given a query Q, with input schema $\mathcal{S}{=}\{R_1, \ldots, R_n\}$, output type $\Gamma{\subseteq}Attr(\mathcal{S})$, and condition set C, with atomic conditions of the form: (i) $A_1\theta a$, (ii) $A_1\theta A_2$, where a is a constant, $A_i{\in}Attr(\mathcal{S})$, for $i{=}1, 2$ and θ is a comparison operator. $Attr$ returns the attributes referred to in the parameter. For a database instance $\mathcal{I}$, the query result $Q[\mathcal{I}]$ is $\pi_\Gamma[\sigma_C[R_1 \times \ldots \times R_n]]$.

We specify a set of tuples missing from $Q[\mathcal{I}]$ by a Why-Not question. This is a query WN having the same input and output schemas as Q and whose conditions in C_W are defined over the projected attributes of Q. Thus, the missing tuples are given by evaluating the query WN over $\mathcal{I}$, i.e., $WN[\mathcal{I}]$. The missing tuples do not appear in the original query result, hence it holds that $Q[\mathcal{I}] \cap WN[\mathcal{I}]{=}\emptyset$.

Compatible tuples play a central role in our setting. Essentially, they are those concatenations of tuples in $\mathcal{I}$ that would yield the missing query results if they were not filtered by Q (e.g, Id_1Id_7 of Ex. 1.1). We define the set CT of compatible tuples by the query eWN that is simply obtained from WN by changing the output schema to $Attr(\mathcal{S})$.

As explained in the introduction, we assume available the *Why-Not polynomial* that, following our definition in (Bidoit et al. 2015a), is a sum of addends, where each addend includes a coefficient and a product of conditions (e.g., $3c_1c_2$, see Ex. 1.1). We refer to each product of query conditions as *explanation*, denoted $\mathcal{E}$ and use PEX to denote the set of all explanations given by the polynomial. In the sequel, we interchangeably present an explanation as a set or a product of conditions. The explanation $\mathcal{E}_\tau$ associated to a given compatible tuple $\tau \in CT$ is defined as $\{c | c{\in}C \text{ and } \tau{\not\models}c\}$. Note that each compatible tuple is pruned by a unique explanation. For instance, $\mathcal{E}_{Id_1Id_7} = \{c_1c_2\}$.

Example 2.1. *The Why-Not question of Ex. 1.1 is expressed by the relational query* $\pi_{name,title}[\sigma_{name='DiCaprio'}[Movie \times Actor]]$. *The set of compatible tuples is returned by* $\sigma_{name='DiCaprio'}[Movie \times Actor]$. *One compatible tuple is* Id_4Id_{10}, *denoting the concatenation of the tuples* Id_4 *and* Id_{10}. *It does not satisfy* c_2, *thus the explanation for its exclusion is* $\mathcal{E}_{Id_4Id_{10}} = \{c_2\}$. *The set of all explanations is* $PEX{=}\{c_2, c_3, c_1c_2, c_2c_3, c_1c_3, c_1c_2c_3\}$ *(Fig. 1(d)).*

Given a query Q, a Why-Not question WN, a database instance $\mathcal{I}$, and the set of explanations PEX, the **query refinement problem** is stated as follows: find queries Q' such that they return (i) all the original result tuples i.e., $Q[\mathcal{I}] \subset Q'[\mathcal{I}]$, and (ii) at least one missing tuple i.e., $Q'[\mathcal{I}] \cap WN[\mathcal{I}] \neq \emptyset$.

A query can be refined in many ways, but *FixTed* uses the explanations in PEX in order to compute *useful* refinements. We quantify the notion of usefulness of a refinement Q' by two metrics: *precision* and *similarity*.

Precision is $\frac{|Q'[\mathcal{I}]| - FP_{Q'}}{|Q'[\mathcal{I}]|}$, where $FP_{Q'}$ is the number of false positive tuples, i.e., tuples output by Q' that neither exist in the result of Q nor correspond to missing tuples.

Similarity. Our approach changes a query Q, by either altering existing conditions or by adding new ones. Essentially, changing a selection condition means adjusting its constant value, whereas this is not applicable for join conditions. As for adding new conditions, we do so to improve the precision. In summary, the similarity of Q' w.r.t. the original query Q, is measured by (i) the number of changed conditions, (ii) the distance of the new constants to the ones of the original conditions (when applicable), and (iii) the number of added conditions.

In our current implementation, the similarity between Q and Q' is a weighted sum (denoted Sim) of these three metrics. Note that we choose difference for the distance of numerical values and edit distance for string values, although any distance measure can be used as a black box. Also, the overall usefulness of a refinement is currently computed as the weighted sum of precision and similarity.

Algorithm 1: *FixTed*

Input: $Q, WN, \mathcal{I}, CT$, *PEX*
Output: SQ: set of query refinements for Q

```
SQ = ∅;
foreach explanation E ∈ PEX do
    E_σ = {c | c ∈ E ∧ c a selection condition};
    E_⋈ = {c | c ∈ E ∧ c a join condition};
    Q_E = {Q};
    if E_σ ≠ ∅ then
        Q_mdr = MinDistRefinement(CT, E_σ, Q); /* Sec. 3.1 */
        Q_E = Q_mdr ∪ Preciser(Q_mdr, CT, Q, I); /* Sec. 3.1 */
    if E_⋈ ≠ ∅ ∧ outerJoinsApply(Q, WN, E_⋈) /* Thm. 3.2 */
    then
        Q_j = JoinToOuterJoin(Q_E, E_⋈); /* Def. 3.4 */
        Q_E = AdjustForSel(Q_j, WN, E); /* end Sec. 3.2 */
    SQ = SQ ∪ Q_E;
return rank(SQ);
```

3. The FixTed Algorithm

We now discuss how *FixTed* computes a set of useful refinements SQ, given a query Q, a database instance $\mathcal{I}$, a Why-Not question WN, the set of compatible tuples CT, and the set of explanations *PEX*. Algorithm 1 shows a high-level pseudo-code. Essentially, *FixTed* iterates over each explanation and first splits the conditions of an explanation into selection and join conditions. It then continues by treating the selection conditions (e.g., c_2) of an explanation. First, Minimum Distance refinement optimizes similarity while relaxing the conditions. Second, the $Preciser$ tackles precision by adding new conditions to eliminate false positives. For join conditions (e.g., c_3), we opt for outer-join rewriting, when possible (as checked by $outerJoinsApply$. As the null values introduced by outer-joins may interfere with the query selections, we adjust the refined query so that the expected results are produced. Finally, refinements are ranked based on their usefulness.

In the remainder of this section, we further discuss the individual steps of the algorithm. Without loss of generality, we assume that an explanation $\mathcal{E}$ either contains only selections or only joins.

3.1 Changing Selections

Minimum Distance Refinement Let $\mathcal{E}$ be an explanation with selection conditions. As multiple compatible tuples may have the same explanation, we refer to them by $CT_{\mathcal{E}} = \{\tau | \mathcal{E}_\tau = \mathcal{E}\}$. Intuitively, we are going to relax the condition(s) of $\mathcal{E}$ s.t. we enable compatible tuples from $CT_{\mathcal{E}}$ to contribute to the expected result. In general, each compatible tuple may lead to one query refinement. However some of them are irrelevant as there similarity to Q is guaranteed not to be minimal. To prune these irrelevant compatible tuples, we resort to computing *skyline* tuples (Borzsony et al. 2001; Papadias et al. 2003) in the set $CT_{\mathcal{E}}$ and only keep tuples on the skyline for further processing. That is, a tuple τ_1 is pruned if there exists another tuple τ_2, whose values are 'closer' to the respective values in $\mathcal{E}$, corresponding to the values of the original query Q.

The tuple value distance for a τ, on an attribute A, is denoted by Δ^A_τ. As we said in Sec. 2, it is either measured by numerical difference or edit distance.

Definition 3.1. *(Skyline tuples) Let $\tau_1, \tau_2 \in CT_{\mathcal{E}}$, given an explanation $\mathcal{E}$ and consider the set of attributes $\mathcal{A}$=$Attr(\mathcal{E})$. We say that τ_1 dominates τ_2 (denoted $\tau_1 \succ_{\mathcal{A}} \tau_2$) if*

1. $\forall A \in \mathcal{A} : \Delta^A_{\tau_1} \leq \Delta^A_{\tau_2}$ *(denoted $\tau_1 \succcurlyeq_{\mathcal{A}} \tau_2$), and*
2. $\exists A \in \mathcal{A} : \Delta^A_{\tau_1} < \Delta^A_{\tau_2}$ *(denoted $\tau_1 \succ_{\mathcal{A}} \tau_2$).*

Then, τ_2 is a skyline *tuple iff there does not exist a tuple τ_1 s.t. $\tau_1 \succ_{\mathcal{A}} \tau_2$. The set of skyline tuples w.r.t. $\mathcal{E}$ is denoted $SL_{\mathcal{E}}$.*

Example 3.1. *For the explanation $\mathcal{E}$=c_1c_2 in Fig. 1, the involved attributes are age and rating, whereas $CT_{c_1c_2}$=$\{\tau_2$:Id_2Id_8, τ_3:Id_3Id_9, τ_4:$Id_1Id_7\}$.*

Fig. 2 displays these tuples in the 2-dimensional space defined by age and rating. For instance, the coordinates of τ_2 result from $\tau_2(age) = 38$ and $\tau_2(rating) = 7.6$. The coordinates of the point marked by $\mathcal{E}$ are given by $\mathcal{E}(age) = 35$ and $\mathcal{E}(rating) = 8.2$. For now, ignore the other points in the diagram. By Def. 3.1, it is easy to see that τ_3 dominates τ_2, whereas τ_3 and τ_4 do not dominate one another. Thus, the skyline tuples for c_1c_2 are $SL_{c_1c_2} = \{\tau_3, \tau_4\}$.

So far, we have considered an explanation in isolation from the other explanations. To improve the efficiency, we further prune compatible tuples not yielding better refinements w.r.t. to similarity using the concept of local skyline tuples. This is done by taking into account the skyline tuples of sub-explanations of $\mathcal{E}$.

Example 3.2. *Consider explanation c_2, having CT_{c_2}=$\{\tau_1$:$Id_4Id_{10}\}$. It is easy to prove that any compatible tuple, e.g., τ_1, cannot be dominated by a compatible tuple pruned by a super-explanation, in our example c_1c_2. However the reverse is possible. For instance, in Fig. 2, we observe that $\tau_1 \succ_{\{age,rating\}} \tau_3$. So, even though τ_3 is a skyline tuple for the explanation c_1c_2 in isolation, we know that there is a more similar refinement requiring less changes to less attribute values. Thus, we can safely prune τ_3.*

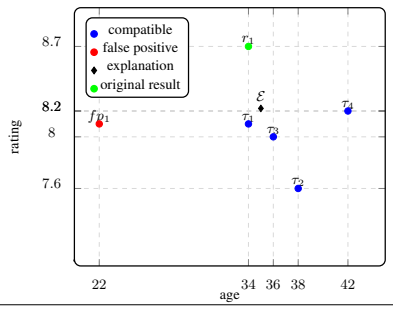

Figure 2. Example for minimum distance refinement

Definition 3.2 (Local skyline tuples w.r.t. an explanation)**.** *Let $SL_{\mathcal{E}}$ be the set of skyline tuples for an explanation $\mathcal{E}$ by Def. 3.1. Let E be the set of sub-explanations of $\mathcal{E}$, and $T = \bigcup_{\mathcal{E}' \in E} SL_{\mathcal{E}'}$ be the union of the skyline tuples for the explanations in E. Then, the set of* local skyline *tuples w.r.t $\mathcal{E}$ is*

$$lSL_{\mathcal{E}} = \{t \in SL_{\mathcal{E}} \mid \nexists t' \in T \; s.t. \; t' \succ_{Attr(\mathcal{E}_{t'})} t\}$$

Each tuple in $lSL_{\mathcal{E}}$ yields one query refinement, obtained by replacing the values of the conditions in $\mathcal{E}$ by the values from the tuples in $lSL_{\mathcal{E}}$. If the comparison in the original condition is $<$ ($>$), then it is changed to $\leq$ ($\geq$). If it is $=$, it is changed to $\leq$ or $\geq$. If it is $\neq$, then the condition is completely removed. We denote by Q_{mdr} the set of query refinements obtained in this phase.

Example 3.3. *From the previous examples, we conclude two sets of local skyline tuples: lSL_{c2}=$\{\tau_1\}$ and lSL_{c_1c2}=$\{\tau_4\}$. This entails two refinements: one (resp. the other) with the effect to get τ_1 (resp. τ_4) appear in the result in addition to the original result tuple r_1. Thus, we obtain the set of queries Q_{mdr}=$\{Q_1, Q_4\}$, with condition sets C_1=$\{c_1, rating \geq 8.1, c_3\}$ and C_4=$\{age \leq 42, rating \geq 8.2, c_3\}$, respectively. Note that Fig. 1(c) corresponds to Q_1.*

Related works (Tran and Chan 2010; Islam et al. 2014) also use the skyline method for selection refinement. However they consider (i) all compatible tuples simultaneously and (ii) all the attributes from the selections in Q, even if these selections are not problematic (e.g., not part of any explanation). This approach renders the computation of the skyline tuples more complex and may lead to changing correct selections in the query. On the other hand, using the Why-Not polynomial we consider subsets of (partial) compatible tuples and subsets of the attributes in selections of Q, and thus we avoid unnecessary tuple comparisons and conditions changes.

As we said, the Why-Not polynomial models all the possible ways in which the conditions of the query are erroneous. For each such explanation, we guarantee to find the refinement with minimal similarity. The similarity is based on three metrics (see Sec. 2). For a given explanation $\mathcal{E}$, the number of conditions is invariant, so the similarity does not depend on the first metric. As for the third metric, adding conditions for the benefit of precision can only reduce similarity. Thus, finding the minimal similarity reduces to finding the minimal value changes in conditions (second metric), which is guaranteed by the procedure described above. This leads to the following theorem.

Theorem 3.1 (Maximal Similarity)**.** *Let Q_m be a query refinement of Q, and let Sim_m be its similarity to Q. If for each $Q_i \in Q_{mdr}$, it holds that $Sim_m \leq Sim_i$, then $Q_m \in Q_{mdr}$.*

Preciser We now move to the phase that aims at improving the precision of refinements in Q_{mdr} (generating potentially less similar, but more precise solutions the user can then choose from). To improve precision, we further constrain the queries in Q_{mdr} by either changing more conditions or by *adding* new conditions. Note that we do not guarantee that the produced refinements include the most precise query refinement. The local skyline tuples again play a central role in generating the new conditions.

Example 3.4. *Consider the query Q_1 associated with the skyline tuple τ_1 in Ex. 3.3. $Q_1[\mathcal{I}]$=$\{r_1, \tau_1, fp_1\}$ contains the false positive tuple fp_1:Id_5Id_{11}. In Fig. 2 we see that changing the other selection condition $age \leq 35$ of Q_1 to $age \leq 34$ (the minimum age to retain both r_1 and τ_1 in the result) does not eliminate fp_1. Thus, we proceed by adding more selections, using attributes from the input query schema. For Q_1 we obtain the following three refinements with a 100% precision (we only display the condition sets):*

$C_1'=\{c_1, rating \geq 8.1, 194 \leq length \leq 136, c_3\}$
$C_1''=\{c_1, rating \geq 8.1, 1997 \leq year \leq 1999, c_3\}$
$C_1'''=\{c_1, rating \geq 8.1, Matrix \leq title \leq Titanic, c_3\}$

3.2 Changing Joins

Applicability of Refining Joins to Outer Joins We now move to processing join conditions. Again, for conciseness and without loss of generality, let us assume that $\mathcal{E}$ includes join conditions only.

In principle, we could employ the same technique as described above. However, this would relax equi-joins to theta joins or even cross products. Indeed, there is no other possibility to make the corresponding compatible tuple, e.g., Id_6Id_{10} participate to the query result. However, cross products are usually not recommended for efficiency reasons typically have low precision scores. Hence, we investigate an alternative resorting to outer joins.

Central to our solution is the notion of *direct* relations, which contain the exact information that the user is missing. For example, $Actor$ is a direct relation because it contains tuples with the missing value $DiCaprio$, whereas $Movie$ is *indirect*.

Definition 3.3 (Direct relation)**.** *Let $R \in \mathcal{S}$ be a relation, and C_W be the condition set of the Why-Not question. Then, R is* direct *iff $\exists c \in C_W$ s.t. c is defined over R. Otherwise, R is* indirect.

A join in an explanation indicates that the compatible tuples from the direct relation do not find the correct join partners. Thus, replacing a join by an outer join may solve the problem.

Example 3.5. *If we replace the join c_3, i.e., $Actor \bowtie_{c_3} Movie$ by $Actor \rtimes\!\!\bowtie_{c_3} Movie$ and assume (for the moment) that no selections exist in the query, we obtain the tuple $(DiCaprio, NULL)$ that satisfies the Why-Not question.*

Unfortunately, it is not always possible to refine a join condition with outer join. Intuitively, this depends on whether we can put all the direct relations as the left (resp. right) operand of the left outer join (resp. right outer join). This is necessary in order to retrieve all missing values specified by the Why-Not question in a single tuple (all other cases replace some missing values with null values, and so fail to satisfy the Why-Not question).

Let G_Q denote the graph modeling the relations (nodes R) and joins (edges c) of Q. Two relations R_1 and R_2 are connected in the graph if there exists in C_Q a join $R_1.A\ \theta\ R_2.B$. The following theorem states that a join refinement is applicable if the join in the explanation does not (transitively) connect two direct relations.

Theorem 3.2. *Let G_Q be the join graph of a query Q and let $\mathcal{E}=\{c_1\}$ be a join explanation, where $c_1 = R_1.A\ \theta\ R_2.B$. Then, a join refinement is applicable for Q, iff c_1 does not belong to any path from relation R_k to R_l in G_Q.*

Join refinement After checking that a join refinement is applicable, we proceed by changing the join to left (or right) outer-join.

Definition 3.4. *(Join refinement) With the above hypothesis, and if applicable by Theorem 3.2, a* join refinement *Q' for Q is given by $Q'=(\mathcal{S}, \Gamma, C')$ s.t. $C'=(C \setminus \{c_1\}) \cup \{R_1 \rtimes\!\!\bowtie_\theta R_2\} (resp. R_1 \bowtie\!\!\ltimes_\theta R_2)$ and R_1 (resp. R_2) is a direct relation.*

Join refinements are easily extended to more than one join in $\mathcal{E}$.

Adjust for selections Up to this point, we have considered the join in isolation. However, if we look back to the example query and the selections c_1 and c_2, the join refinement of Ex. 3.5 would not return $DiCaprio$, as the NULL values introduced by the outer join fail the selection predicate of c_2. We need to adjust our final refinement. We distinguish two cases: if the schema constraints prevent the source data to contain NULL values in attributes of a selection c_i of the form `R.A θ v`, where R is an indirect relation, then the condition may be rewritten to (c_i `OR R.A IS NULL`). Otherwise, we create subqueries for replacing the relations over which selections are specified. Then, the joins of the query are specified over the respective subqueries in the refined query.

Example 3.6. *Considering the full query of our running example and the explanation c_3, the refined query corresponds to the following SQL query statement.*

```
SELECT A.name, M.title
FROM (SELECT * FROM Actor WHERE age ≤ 35) AS A
LEFT OUTER JOIN
    (SELECT * FROM MOVIE WHERE rating > 8.2) AS M
ON A.film = M.title
```

4. Outlook and Future Work

This paper presents *FixTed*, the first algorithm leveraging Why-Not polynomials to refine queries such that missing, but expected query results are recovered. *FixTed* thereby improves on the state of the art by guaranteeing refinements with maximum similarity to the original query and ensuring that all possible cases of why the desired data was missing are covered. Additionally, it is the first algorithm to refine joins using outer joins. A prototype implementation (Bidoit et al. 2015b) demonstrates the feasibility and effectiveness of our approach. Besides a thorough experimental evaluation of *FixTed*, we plan to extend it in several aspects,e.g, by introducing more refinement possibilities for joins.

References

N. Bidoit, M. Herschel, and K. Tzompanaki. Immutably answering why-not questions for equivalent conjunctive queries. In *USENIX TAPP Workshop*, 2014.

N. Bidoit, M. Herschel, and K. Tzompanaki. Efficient computation of polynomial explanations of why-not questions. In *CIKM*, 2015a.

N. Bidoit, M. Herschel, and K. Tzompanaki. EFQ: Why-not answer polynomials in action. *PVLDB*, 8(12):1980–1983, 2015b.

S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430. IEEE, 2001.

L. Chen, X. Lin, H. Hu, C. S. Jensen, and J. Xu. Answering why-not questions on spatial keyword top-k queries. In *ICDE*, pages 279–290, 2015.

T. Gaasterland, P. Godfrey, and J. Minker. Relaxation as a platform for cooperative answering. *Journal of Intelligent Information Systems*, 1992.

Y. Gao, Q. Liu, G. Chen, B. Zheng, and L. Zhou. Answering why-not questions on reverse top-k queries. *PVLDB*, 8(7):738–749, 2015.

Z. He and E. Lo. Answering why-not questions on top-k queries. *IEEE TKDE*, 26(6):1300–1315, 2014.

J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVDLB*, 1(1), 2008.

M. S. Islam, C. Liu, and R. Zhou. User feedback based query refinement by exploiting skyline operator. In *ER*, pages 423–438, 2012.

M. S. Islam, R. Zhou, and C. Liu. On answering why-not questions in reverse skyline queries. In *ICDE*, 2013.

M. S. Islam, C. Liu, and R. Zhou. Flexiq: A flexible interactive querying framework by exploiting the skyline operator. *Journal of Systems and Software*, 97:97–117, 2014.

D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478. ACM, 2003.

B. ten Cate, C. Civili, E. Sherkhonov, and W. Tan. High-level why-not explanations using ontologies. In *PODS*, 2015.

Q. T. Tran and C.-Y. Chan. How to ConQueR why-not questions. In *SIGMOD*, 2010.

Q. T. Tran, C. Y. Chan, and S. Parthasarathy. Query reverse engineering. *VLDB Journal*, 23(5):721–746, 2014.