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-Télécop 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.

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 \( \{ \{ \text{Reeves}, \text{Matrix} \} \} \), 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 \( c_1 c_2 \). This entails that the operator \( c_1 \) is now 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 \( Id_1, Id_7 \), \( Id_2, Id_6 \), and \( Id_3, Id_9 \). 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.
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.

- **Query**
  - User query \(Q = (S, T, C')\)
  - Why-Not question \(WN = (S, T, C'_{w})\)
  - Extended Why-Not question \(eWN(S, Attr(S), C_{w})\)

- **Query result**
  - Answer tuples: \(Q[I]\)
  - Missing tuples: \(WN[I]\)
  - Compatible tuples (\(CT\)): \(eWN[I]\)

### Table 1. Preliminary queries and tuples

<table>
<thead>
<tr>
<th>User query (Q = (S, T, C'))</th>
<th>Why-Not question (WN = (S, T, C'_{w}))</th>
<th>Extended Why-Not question (eWN(S, Attr(S), C_{w}))</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Query</strong></td>
<td><strong>Query result</strong></td>
<td><strong>Query result</strong></td>
</tr>
<tr>
<td>User query (Q = (S, T, C'))</td>
<td>Answer tuples: (Q[I])</td>
<td>Missing tuples: (WN[I])</td>
</tr>
<tr>
<td>Why-Not question (WN = (S, T, C'_{w}))</td>
<td><strong>Query result</strong></td>
<td><strong>Query result</strong></td>
</tr>
<tr>
<td>Extended Why-Not question (eWN(S, Attr(S), C_{w}))</td>
<td>Compatible tuples ((CT)): (eWN[I])</td>
<td></td>
</tr>
</tbody>
</table>

**In addition**, \(WN[I]\) returns a comprehensive set of refinements. FixTed ensures that **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.

### 3. The FixTed Algorithm

We now discuss how FixTed computes a set of useful refinements \(SQ\), given a query \(Q\), a database instance \(I\), 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. Lastly, refinements are ranked based on their usefulness.

Pression is \(\left|Q[I]\right| - FPQ\), where \(FPQ\) 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.
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 $\mathcal{CT}_E = \{t | t \in \mathcal{E}, E = \mathcal{E}_t\}$. Intuitively, we are going to relax the condition(s) of $\mathcal{E}$ s.t. we enable compatible tuples from $\mathcal{CT}_E$ 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 $\mathcal{CT}_E$ and only keep tuples on the skyline for further processing. That is, a tuple $t_3$ is pruned if there exists another tuple $t_2$, whose values are ‘closer’ to the respective values in $E$, corresponding to the values of the original query $Q$.

The tuple value distance for a $\tau$, on an attribute $A$, is denoted by $\Delta^A_\tau$. As we said in Sec. 2, it is either measured by numerical difference or edit distance.

**Definition 3.1.** (Skyline tuples) Let $t_1, t_2 \in \mathcal{CT}_E$, given an explanation $\mathcal{E}$ and consider the set of attributes $A := \text{Attr}(\mathcal{E})$. We say that $t_1$ dominates $t_2$ (denoted $t_1 \succ t_2$) if

1. $\forall A \in A : \Delta^A_{t_1} \leq \Delta^A_{t_2}$ (denoted $t_1 \succeq t_2$), and
2. $\exists A \in A : \Delta^A_{t_1} < \Delta^A_{t_2}$ (denoted $t_1 \succ t_2$).

Then, $t_2$ is a skyline tuple iff there does not exist a tuple $t_1 \succ t_2$. The set of skyline tuples w.r.t. $\mathcal{E}$ is denoted $SL_E$.

**Example 3.1.** For the explanation $\mathcal{E} = \{c_1, c_2\}$ in Fig. 1, the involved attributes are age and rating, whereas $\mathcal{CT}_{c_1c_2} = \{t_2: I_d 2 | d_1, t_3: I_d 3 | d_1, t_4: I_d 1 | d_1\}$.

Fig. 2 displays these tuples in the 2-dimensional space defined by age and rating. For instance, the coordinates of $t_2$ result from $t_2(\text{age}) = 38$ and $t_2(\text{rating}) = 7.6$. The coordinates of the point marked by $\mathcal{E}$ are given by $\mathcal{E}(\text{age}) = 35$ and $\mathcal{E}(\text{rating}) = 8.2$. For now, ignore the other points in the diagram. By Def. 3.1, it is easy to see that $t_3$ dominates $t_2$, whereas $t_3$ and $t_4$ do not dominate one another. Thus, the skyline tuples for $c_1c_2$ are $SL_{c_1c_2} = \{t_3, t_4\}$.

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 $\mathcal{CT}_{c_2} = \{t_1: I_d 1 | d_{10}\}$. It is easy to prove that any compatible tuple, e.g., $t_1$, 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 $t_1 \succ (\text{age} \succ \text{rating}) \succ t_3$. So, even though $t_3$ 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 attributes values. Thus, we can safely prune $t_3$.

**Definition 3.2.** (Local skyline tuples w.r.t. an explanation). Let $SL_e$ be the set of skyline tuples for an explanation $\mathcal{E}$ by Def. 3.1. Let $\mathcal{E}$ be the set of sub-explanations of $\mathcal{E}$, and $T = \bigcup_{\mathcal{E} \in T} SL_e$ be the union of the skyline tuples for the explanations in $\mathcal{E}$. Then, the set of local skyline tuples w.r.t. $\mathcal{E}$ is

$$\text{ISL}_E = \{ t \in SL_E | \forall t' \text{ s.t. } t' \succ \text{Attr}(\mathcal{E}) \text{, } t \}$$

Each tuple in $\text{ISL}_E$ yields one query refinement, obtained by replacing the values of the conditions in $\mathcal{E}$ by the values from the tuples in $\text{ISL}_E$. If the comparison in the original condition is $\prec (<)$, then it is changed to $\leq (\geq)$. If it is $\succ$, it is changed to $\geq$ or $\leq$. If it is $\equiv$, 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: $\text{ISL}_{c_1c_2} = \{t_3\}$ and $\text{ISL}_{c_1c_2} = \{t_4\}$. This entails two refinements: one (resp. the other) with the effect to get $t_1$ (resp. $t_3$) appear in the result in addition to the original result tuple $t$. Thus, we obtain the set of queries $Q_{mdr} = \{Q_1, Q_4\}$, with condition sets $C_1 = \{c_1, \text{age} \geq 35, \text{rating} \geq 8.1, \text{c}_2 \}$ and $C_4 = \{\text{age} \leq 42, \text{rating} \geq 8.2, \text{c}_3 \}$, 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 $\text{Sim}_m$ be its similarity to $Q$. If for each $Q_i \in Q_{mdr}$, it holds that $\text{Sim}_m \leq \text{Sim}_i$, then $Q_m \in Q_{mdr}$.

Precisely, 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 $t_1$ in Ex. 3.3. $Q_1[\mathcal{Z} = \{t_1, t_1, f_p\}]$ contains the false positive tuple $f_p: I_d 2 | d_1, 1$. In Fig. 2 we see that changing the other selection condition $\text{age} \leq 35$ of $Q_1$, to $\text{age} \leq 34$ (the minimum age to retain both $t_1$ and $t_1$ in the result) does not eliminate $f_p$. 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):
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 $E$ 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., $Id_{A}Id_{10}$ 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 and the selection join in isolation. However, if we look back to the example query we investigate an alternative resorting to outer joins.

Definition 3.3 (Direct relation). Let $R \sigma S$ 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 $\bowtie_{c_3}$ 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 $G_Q$ a join $R_1 \bowcap_{\theta} R_2$. The following theorem states that a join refinement is applicable if the join in the explanation does not (transitivity) connect two direct relations.

Theorem 3.2. Let $G_Q$ be the join graph of a query $Q$ and let $E=\{c_1\}$ be a join explanation, where $c_1 = R_1 \bowcap_{\theta} R_2$. Then, a join refinement is applicable for $Q$, iff $c_1$ does not belong to any path from relation $R_1$ to $R_2$ 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 $Q'=(S, T, C_1')$, s.t. $C_1' = (C_1 \setminus \{c_1\}) \cup \{R_1 \bowcap_{\theta} R_2\}$ (resp. $R_1 \bowcap_{\theta} R_2$) and $R_1$ (resp. $R_2$) is a direct relation.

Join refinements are easily extended to more than one join in $E$.

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 $c_3$, 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 $c_1$ of the form $R, \bowcap_{\theta} \psi$, where $R$ is an indirect relation, then the condition may be rewritten to ($c_1$ 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. 2015a) 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


