EFQ: Why-Not Answer Polynomials in Action
Nicole Bidoit, Melanie Herschel, Katerina Tzompanaki

To cite this version:
Nicole Bidoit, Melanie Herschel, Katerina Tzompanaki. EFQ: Why-Not Answer Polynomials in Action. 41st International Conference on Very Large Data Bases - VLDB 2015, Aug 2015, Hawai, United States. <hal-01182115>

HAL Id: hal-01182115
https://hal.archives-ouvertes.fr/hal-01182115
Submitted on 31 Jul 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
EFQ: Why-Not Answer Polynomials in Action

Nicole Bidoit
Université Paris Sud / Inria
91405 Orsay Cedex, France
nicole.bidoit@lri.fr

Melanie Herschel
Universität Stuttgart
70569 Stuttgart, Germany
melanie.herschel@ipvs.uni-stuttgart.de

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

ABSTRACT

One important issue in modern database applications is supporting the user with efficient tools to debug and fix queries because such tasks are both time and skill demanding. One particular problem is known as Why-Not question and focusses on the reasons for missing tuples from query results. The EFQ platform demonstrated here has been designed in this context to efficiently leverage Why-Not Answers polynomials, a novel approach that provides the user with complete explanations to Why-Not questions and allows for automatic, relevant query refinements.

1. INTRODUCTION

In the last few years, the research community has been quite active in the field of answering Why-Not questions. From knowledge bases [6] to keyword search systems [1] or commercially interesting reverse top-k queries [8], the different application domains of Why-Not questions are numerous, revealing the importance of understanding why queries do not return all expected results.

This demonstration focuses on the problem of answering Why-Not questions over relational databases and fixing the queries accordingly. To illustrate this problem, consider the database instance and the sample query Q shown in Figure 1. For ease of reference, we have labeled the conditions in Q according to c1, c2 and c3. It is easy to verify that the result of Q is (name: The Godfather, year: 1972). Let us now assume that the user wonders “Why is there not any movie after 2000 in the result?”. The possible reasons for these missing answers abound: are there no other recent movies (i.e., incomplete source data)? Or are the conditions of Q too restrictive (i.e., problematic query)? Or both?

To find an explanation, a developer (or, more generally, a database or a simple internet user) would traditionally try to find the problematic parts of the query by relaxing the restrictions one by one. When finally she would identify a possible explanation, she would move on to fixing the problem. This means that she would test (possibly numerous) different modifications until reaching a new refined query that satisfies her. Obviously, this process is time consuming, it requires skills from the user and, when in a working environment, it can lead to money loss.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vlldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.

Figure 1: Example query and data

<table>
<thead>
<tr>
<th>Movie</th>
<th>Rating</th>
<th>Votes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Australia</td>
<td>8.5</td>
<td>4</td>
</tr>
<tr>
<td>Avatar</td>
<td>9.2</td>
<td>8</td>
</tr>
<tr>
<td>The Godfather</td>
<td>1972</td>
<td></td>
</tr>
</tbody>
</table>

To aid the user in this task and save company resources, we have designed the Explain and Fix Query platform (EFQ). In EFQ, the user is able to execute queries, express a Why-Not question and ask for (1) explanations to Why-Not questions and (2) query refinements that produce the desired results.

Explanations to Why-Not questions. Algorithms computing explanations to Why-Not questions have already been proposed (see [17, 11, 13] for data-based explanations, [7, 3] for query-based explanations, and [9] for explanations considering both simultaneously). As demonstrated in [10], query-based explanations are typically more useful in practice than data-based explanations due to better scalability to large volumes of data. Hence, we concentrate our work here on generating and leveraging query-based explanations. However, algorithms [3, 7] that compute query-based explanations are based on traversing a single query tree of a query Q and as a result produce possibly incomplete explanations. The meaning of incomplete is twofold here: either one explanation misses some parts or more explanations than those computed exist.

As an example, consider again the instance and the query Q shown in Figure 1 and the Why-Not question concerning movies more recent than 2000. One query-based explanation is that the rating of the eligible (i.e., recent enough) movies (m1 and m2 in the Movies table) is too low, thus failing condition c2 of Q. Another valid query-based explanation is the violation of condition c3, i.e., votes<5. Any of [3, 7] may return one of these conditions as explanation. However, the complete explanation for pruning m1 or m2 is that both conditions c2 and c3 are not satisfied (assuming the ‘joined’ tuples m1r1 and m2r2). This information is crucial when thinking of the subsequent fixing phase, as changing any of f these conditions alone will not solve the problem.

To overcome the shortcoming of incomplete query-based explanations, we have recently introduced Why-Not Answer polynomials [2]. These capture both the conjunctive responsibility of conditions (using multiplication) and all possible condition combinations explaining the missing tuples (using addition). The coefficients of each term in the polynomial indicate how many eligible tuple combinations (like the tuple m1r1) are pruned by the specific condition

SELECT M.name, M.year
FROM Movies M,
Ratings R
WHERE M.name = R.movie
AND rating > 9(r2)
AND votes > 5(r3)
WHERE M.name = R.movie(c1)
AND rating < 9(r2)
AND votes < 5(r3)
combination. For instance, the discussion of our example above yields the addend $2c_2c_3$. There are further explanations involving the join as well and the complete Why-Not Answer polynomial is $2c_1 + 2c_2c_3 + 2c_1c_2c_3$.

EFQ relies on the proposed Why-Not answer polynomials to represent all possible reasons for not obtaining the expected results. More specifically in EFQ, users can easily inspect the different reasons for missing tuples with the certainty that any reason (i.e., addend) is complete. They can then select among the explanations which one(s) should be considered in the subsequent refinement phase with the guarantee that the computed refinement will yield missing tuples. To assist the user with this choice, the explanations are ranked based on how many conditions must be changed in order to fix the query. Besides this, we provide an upper bound of the missing tuples that can be retrieved when changing an explanation. This can be useful in cases when the user is interested in the number of missing tuples that can be retrieved. The information necessary to compute these heuristics is easily extracted from the polynomial using the size of addends and coefficients.

Query refinements. When the user does not want or does not know how to refine the query herself, she can rely on the automatic query refinements proposed in our platform. To compute the alternative refinements EFQ exploits the obtained explanations. Actually the user can either select the explanation or rely on the whole set of explanations to obtain query refinements. As query refinements are usually numerous, we provide a cost model for ranking them based on several criteria, such as their edit distance from the original query or the irrelevant results they produce.

Demonstration contribution. The first contribution of EFQ is to demonstrate that Why-Not answer polynomials as defined in [2] can be effectively and efficiently used as query-based explanations and that they form a solid basis for subsequent query refinement. The demonstration features the latest and most efficient algorithms we have defined for generating Why-Not answer polynomials [4] and proposing query refinements. The demonstrated line of work falls within the Nautilus project (http://nautilus-system.org), where previous demonstrations [12, 10] concentrated on data-based explanations or a comparison of data- and query-based explanations. This demonstration is the first to focus on the benefits of Why-Not answer polynomials for query-based explanations as well as on the query refinement step that leverages these rich explanations.

2. EFQ PLATFORM ARCHITECTURE

EFQ is a platform that facilitates the query debugging and fixing experience 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 2.

There are three basic components in EFQ: (1) the Scenario component, (2) the Explanation component and, (3) the Refinement component. EFQ relies on a database management system and provides an interface for the user-platform interactions.

In the following, we will discuss the three main components in more detail. For illustration, we will reuse the movie example already used in the introduction.

2.1 Scenario Component

Firstly, a query $Q$ is specified over a selected database instance $I$. In our setting, we allow conjunctive queries with inequalities. Note that extending our framework to union of conjunctive queries or count aggregate queries is trivial. However negation and other aggregation functions need further investigation. Then, a Why-Not question $WN$ is specified for the missing tuples from the result of $Q$.

For example, consider again the result of the query $Q$ in Figure 1, which is $(\text{name} \land \text{The Godfather}, \text{year}\geq 1972)$. The user asks ‘Why is there not any movie after 2000?’ This can be captured as $WN=_{\text{name} = ? \land \text{year} > 2000}$. Looking at the database instance in Figure 1, we easily find that $WN$ corresponds to two missing tuples ($\text{Australia, 2008}$) and ($\text{Aviator, 2009}$).

The main task of the Scenario component is to identify tuples built from $I$ and from which we can get the missing tuples. For this, we rely both on $Q$ and $WN$. We consider missing tuples as projections of tuples satisfying $WN$ and typed by the FROM clause schema of $Q$. In our example, the tuples $m_1 r_1$, $m_2 r_2$ or $m_3 r_3$ defined over $\text{M.name}, \text{M.year}, \text{R.movie}, \text{R.rating}$ and $\text{R.votes}$ lead to the missing tuple ($\text{Australia, 2008}$) by projection over $\text{M.name}$ and $\text{M.year}$. In the same spirit, the missing tuple ($\text{Aviator, 2009}$) results from $m_2 r_1$, $m_2 r_2$ or $m_3 r_3$. The tuples $m_1 r_1, \ldots, m_2 r_3$ leading to the missing ones are called compatible. Next, the set of compatible tuples is denoted by $CT$.

If $CT$ is empty, there are no data in $I$ that match the Why-Not question and thus neither query-based explanations nor query refinements can be expected. Details on the definition and identification of compatible tuples can be found in [2].

2.2 Explanations Component

To compute the query-based explanations, the Explanation component relies on the polynomial formalisation of a Why-Not answer (a.k.a. answer to the Why-Not question) introduced in [2]. This formalisation builds on the set $CT$ of compatible tuples.

Each compatible tuple in $CT$ has been eliminated by one or more conditions of the query $Q$. Intuitively, finding these conditions provides one explanation for missing a tuple.

Take for example the compatible tuple $m_1 r_1$. It is easy to see that this tuple does not satisfy the conditions votes $\geq 5$ and rating $\geq 9$. In other words, it was pruned out because of these conditions, and they together form an explanation.

Finding all the possible explanations amounts to finding the explanation for each compatible tuple in $CT$. Then, summing up all explanations leads to a polynomial-like representation of the Why-Not answer. In this polynomial, the variables are conditions of the

1. Next, if $m$ and $r$ are two tuples, $m r$ denotes their concatenation.
query \( Q \) and the coefficient of a term is the number of compatible tuples pruned by this term (i.e., explanation).

In our example, there are three different explanations (recall conditions have been named \( c_i \)). \( c_2c_3 \) prunes two compatible tuples, \( c_1 \) prunes another two and \( c_1c_2c_3 \) the last two. We thus obtain the polynomial \( 2c_1 + 2c_2c_3 + 2c_1c_2c_3 \) already given in the introduction.

The naive presentation of the polynomial answer of a Why-Not question does not directly provide an efficient way of computing it. Instead, the platform \( EFQ \) relies on the development and implementation of an efficient algorithm described in [4]. This algorithm uses data partitioning based both on the conditions of the Why-Not question and of the query \( Q \). Instead of iterating over the set of compatible tuples, it iterates over the power set of the condition set of \( Q \), which is normally much smaller. Moreover, and whenever possible, it mathematically computes the number of compatible tuples eliminated by each explanation, in order to avoid costly database operations.

### 2.3 Refinement Component

The goal now is to produce alternative queries, so-called refinements of the original query, whose result includes at least one of the missing tuples as specified by \( WN \). These query refinements can be computed in two ways: based on a selected explanation or on the whole set of explanations. Explanations and compatible tuples are essential for this phase.

In the literature, we find algorithms to compute query relaxations [15, 16] or query reformulations [18] to retrieve missing tuples. Our query refinement algorithm builds on ideas in [18], appearing also in [14]. These algorithms use the notion of skyline [5] to prune the space of refined queries. However, neither [18] nor [14] uses any knowledge of what is wrong with the query, they solely rely on the set of compatible data. On the contrary, our algorithm takes into account the explanations provided by the previous step. In this way, we can (1) provide ‘targeted’ refinements, i.e., refinements altering specific explanations, (2) create maximum similarity refinements, by making the minimum possible changes in the query conditions, and (3) faster compute the refinements, by introducing the notion of local (w.r.t. the explanation) skyline tuples.

The algorithm computing the refinements proceeds in two steps. In the first step, for each explanation, refinements are computed avoiding those that are for sure less similar to \( Q \) than others. In order to do this, we rely on the set of skyline compatible tuples for each explanation.

For example, let us consider the explanation \( c_2c_3 \). The two compatible tuples it eliminates are \( m_1r_1 \) and \( m_2r_2 \). \( m_1r_1 \) has higher constant values for the attributes \( rating \) and \( votes \), thus is closer to the user’s initial intention. Subsequently, the refined query based on \( c_2c_3 \) is generated based on \( m_1r_1 \) and has the condition part \( name = movie \ AND rating >= 8.5 \ AND votes >= 4 \).

The second step of the algorithm focuses on removing false positive tuples (tuples neither produced by the original query nor by the Why-Not question) from the queries generated by the first step. In addition to changing already existing conditions in the query, this step may introduce in the refined queries also new conditions.
3. USER INTERFACE

EFQ provides an interface for the user-system interaction, exposed in Figure 3. In this section, we briefly walk the reader through a sample interaction with our platform.

In the home page (Figure 3(a)) the user can choose a database and provide a query by selecting an existing view or by writing an SQL statement in the provided text area. After submitting the query, the upper part of the interface displays the results and the original query (Figure 3(b)). Then, the user specifies the Why-Not question by inserting conditions on the attributes in the provided table. If there exist compatible tuples for the Scenario component to identify, the interface allows the user to continue and ask either for explanations (debug query) or for query refinements (fix query).

Figure 3(c) displays the interface for the Explanation component. The user can navigate in this page through the existing explanations, ordered by (1) size or (2) selectivity. The selectivity metric for a given explanation is calculated based on the number of compatible tuples pruned by the conditions in the explanation. As such, it provides the user with an upper bound on the number of missing tuples that may be recovered from repairing this explanation. As the user navigates through the existing explanations, the upper part of the interface highlights the corresponding problematic parts of the query. At any given point, the user is able to ask for query refinements using the current explanation, or she can ask for query refinements based on all explanations, by clicking the ‘Query fixing’ menu tab.

Figure 3(d) demonstrates the interface of the Refinement component. 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 (i.e., irrelevant) tuples appear in the result of the new query. Furthermore, she 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. This is coupled with highlighting, on the original query, the explanation used to generate the given refinement\(^1\), at the upper part of the interface.

4. DEMONSTRATION

The EFQ platform is implemented in Java and JSP, and is currently offered as an application connected to a PostgreSQL DBMS. At the conference the attendees will learn how to use the platform, through a number of predefined scenarios and at least three use cases on different domains.


\(^1\)Each highlighted condition in the original query is associated with a highlighted condition in the refined one (but not vice-versa).

5. REFERENCES