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 as c_1, c_2 and c_3. 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.

Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

```
SELECT M.name, M.year
FROM Movies M,
     Ratings R
WHERE M.name = R.movie(c1)
AND rating > 9(c2)
AND votes > 5(c3)
```

Movies

	name	year
m_1	*Australia*	2008
m_2	*Avatar*	2009
m_3	*The Godfather*	1972

Ratings

	movie	rating	votes
r_1	*Australia*	7	3
r_2	*Avatar*	8.5	4
r_3	*The Godfather*	9.2	8

Figure 1: Example query and data

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 (m_1 and m_2 in the Movies table) is too low, thus failing condition c_2 of Q. Another valid query-based explanation is the violation of condition c_3, i.e., $votes{>}5$. Any of [3, 7] may return one of these conditions as explanation. However, the *complete* explanation for pruning m_1 or m_2 is that both conditions c_2 and c_3 are not satisfied (assuming the ‘joined’ tuples m_1r_1 and m_2r_2). 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 m_1r_1) are pruned by the specific condition 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 certitude 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 $\mathcal{I}$. 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

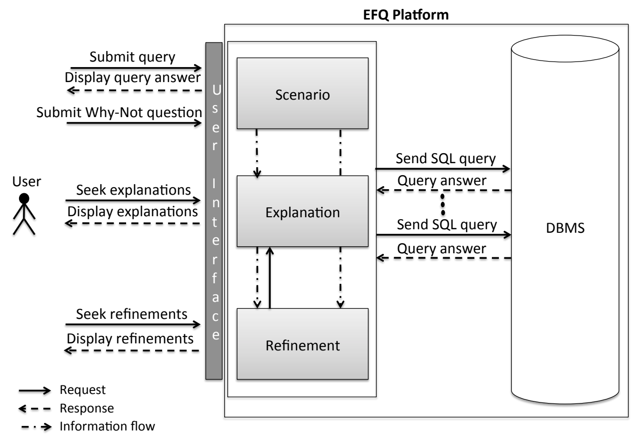


Figure 2: *EFQ* platform overview

question $\mathcal{WN}$ 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 $(name{:}The\ Godfather, year{:}1972)$. The user asks 'Why is there not any movie after 2000?' This can be captured as $\mathcal{WN}{=}(name{:}?, year{:}{>}2000)$. Looking at the database instance in Figure 1, we easily find that $\mathcal{WN}$ corresponds to two missing tuples $(Australia, 2008)$ and $(Avatar, 2009)$.

The main task of the Scenario component is to identify tuples built from $\mathcal{I}$ and from which we can get the missing tuples. For this, we rely both on Q and $\mathcal{WN}$. We consider missing tuples as projections of tuples satisfying $\mathcal{WN}$ and typed by the $FROM$ clause schema of Q. In our example, the tuples[1] m_1r_1, m_1r_2 or m_1r_3 defined over M.name, M.year, R.movie, R.rating and R.votes lead to the missing tuple $(Australia, 2008)$ by projection over M.name and M.year. In the same spirit, the missing tuple $(Avatar, 2009)$ results from m_2r_1, m_2r_2 or m_2r_3. The tuples m_1r_1, …, m_2r_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 $\mathcal{I}$ 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

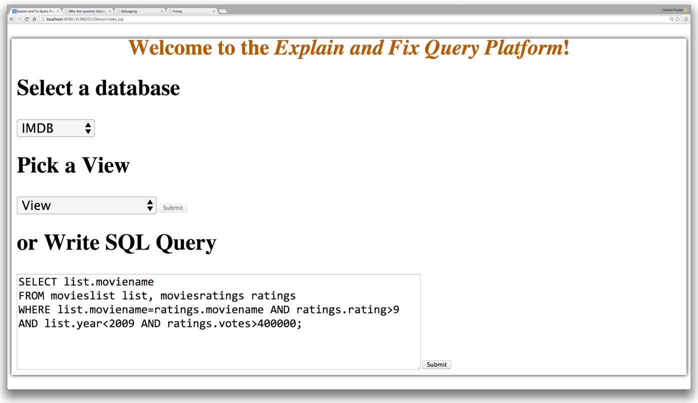
To compute the query-based explanations, the Explanation component relies on the polynomial formalisation of a *Why-Not answer* (a.k.a. answer to the Why-Not question) introduced in [2]. This formalisation builds on the set CT of compatible tuples .

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_1r_1. It is easy to see that this tuple does not satisfy the conditions $votes{>}5$ and $rating{>}9$. 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, mr denotes their concatenation.



Welcome to the *Explain and Fix Query Platform*!

Select a database

IMDB

Pick a View

View

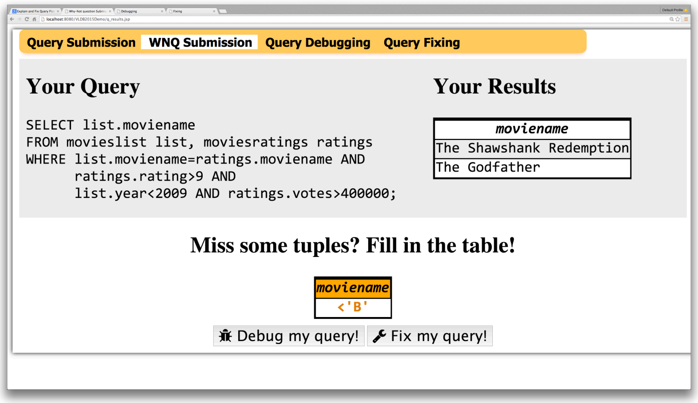
Submit

or Write SQL Query

```
SELECT list.moviename
FROM movieslist list, moviesratings ratings
WHERE list.moviename=ratings.moviename AND ratings.rating>9
AND list.year<2009 AND ratings.votes>400000;
```

Submit

(a) User defines query



Your Query

```
SELECT list.moviename
FROM movieslist list, moviesratings ratings
WHERE list.moviename=ratings.moviename AND
      ratings.rating>9 AND
      list.year<2009 AND ratings.votes>400000;
```

Your Results

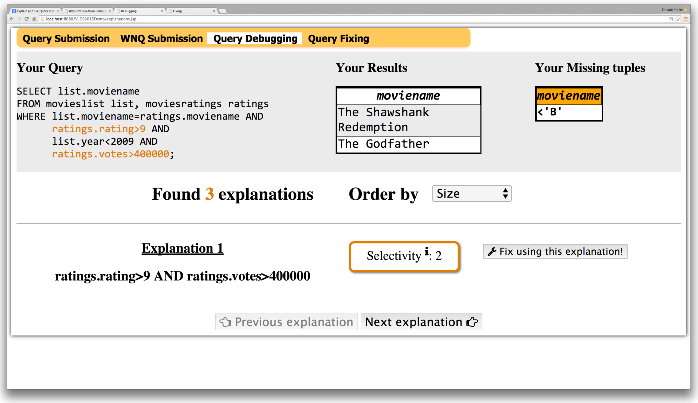
moviename
The Shawshank Redemption
The Godfather

Miss some tuples? Fill in the table!

moviename
<'B'

Debug my query! Fix my query!

(b) User defines Why-Not question



Your Query

```
SELECT list.moviename
FROM movieslist list, moviesratings ratings
WHERE list.moviename=ratings.moviename AND
      ratings.rating>9 AND
      list.year<2009 AND
      ratings.votes>400000;
```

Your Results

moviename
The Shawshank Redemption
The Godfather

Your Missing tuples

moviename
<'B'

Found 3 explanations Order by Size

Explanation 1

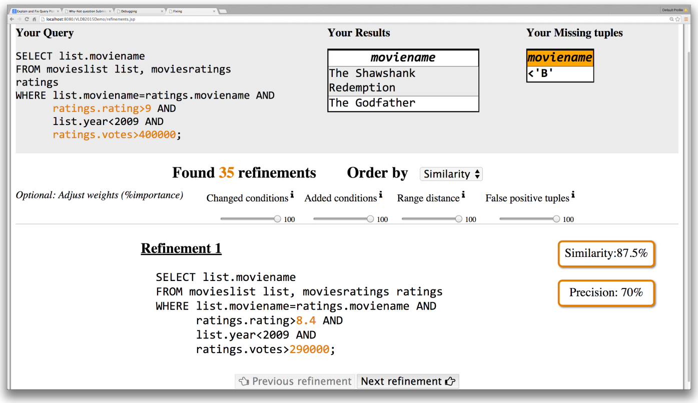
ratings.rating>9 AND ratings.votes>400000

Selectivity: 2

Fix using this explanation!

Previous explanation Next explanation

(c) User navigates through explanations



Your Query

```
SELECT list.moviename
FROM movieslist list, moviesratings
ratings
WHERE list.moviename=ratings.moviename AND
      ratings.rating>9 AND
      list.year<2009 AND
      ratings.votes>400000;
```

Your Results

moviename
The Shawshank Redemption
The Godfather

Your Missing tuples

moviename
<'B'

Found 35 refinements Order by Similarity

Optional: Adjust weights (%importance) Changed conditions 100 Added conditions 100 Range distance 100 False positive tuples 100

Refinement 1

```
SELECT list.moviename
FROM movieslist list, moviesratings ratings
WHERE list.moviename=ratings.moviename AND
      ratings.rating>8.4 AND
      list.year<2009 AND
      ratings.votes>290000;
```

Similarity:87.5%

Precision: 70%

Previous refinement Next refinement

(d) User navigates through query refinements

Figure 3: *EFQ* Interface

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 $\mathcal{WN}$. 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[2], 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.

The first use case is based on the IMDB database (http://www.imdb.com), which contains information about movies. The second use case is about financial activities of american congressmen stored in the Congress database (and gathered at http://bioguide.congress.gov, http://usaspending.gov, http://earmarks.omb.gov). The third use case relies on data and queries about products and orders generated by the TPC-H benchmark (http://www.tpc.org/tpch/).

The first two use cases describe real world scenarios, whereas the third one is based on synthetic data. To demonstrate the behaviour of our platform, we vary the input parameters in the predefined scenarios. More specifically, we take into account simple and complex (1) queries and (2) Why-Not questions, by changing the number and type of associated conditions, and (3) different database sizes. To support the usefulness of *EFQ*, we show that knowing the complete explanations not only provides the user with alternative debugging options, but also prevents him from (possibly numerous) pointless debugging attempts. Furthermore, we show the possibilities offered to the user to obtain targeted or general refinements.

[2]Each highlighted condition in the original query is associated with a highlighted condition in the refined one (but not vice-versa).

5. REFERENCES

[1] A. Baid, W. Wu, C. Sun, A. Doan, and J. F. Naughton. On debugging Non-Answers in keyword search systems. In *EDBT*, 2015.

[2] N. Bidoit, M. Herschel, and K. Tzompanaki. Immutably answering Why-Not questions for equivalent conjunctive queries. In *TAPP*, 2014.

[3] N. Bidoit, M. Herschel, and K. Tzompanaki. Query-based Why-Not provenance with Nedexplain. In *EDBT*, pages 145–156, 2014.

[4] N. Bidoit, M. Herschel, and K. Tzompanaki. Efficiently and Effectively Answering Why-Not Questions based on Provenance Polynomials. Research Report RR-8697, 2015.

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

[6] H. Chalupsky and T. A. Russ. WhyNot: debugging failed queries in large knowledge bases. In *AAAI/IAAI*, pages 870–877, 2002.

[7] A. Chapman and H. V. Jagadish. Why not? In *SIGMOD*, pages 523–534, 2009.

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

[9] M. Herschel. A hybrid approach to answering Why-Not questions on relational query results. *JDIQ*, pages 10:1–10:29, 2015.

[10] M. Herschel and H. Eichelberger. The Nautilus Analyzer: understanding and debugging data transformations. In *CIKM*, pages 2731–2733, 2012.

[11] M. Herschel and M. A. Hernández. Explaining missing answers to SPJUA queries. *PVLDB*, pages 185–196, 2010.

[12] M. Herschel, M. A. Hernández, and W. C. Tan. Artemis: A system for analyzing missing answers. *PVLDB*, pages 1550–1553, 2009.

[13] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, pages 736–747, 2008.

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

[15] C. Mishra and N. Koudas. Interactive query refinement. In *EDBT*, pages 862–873, 2009.

[16] D. Mottin, A. Marascu, S. B. Roy, G. Das, T. Palpanas, and Y. Velegrakis. A probabilistic optimization framework for the empty-answer problem. *PVLDB*, pages 1762–1773, 2013.

[17] S. Riddle, S. Köhler, and B. Ludäscher. Towards constraint provenance games. In *TaPP*, 2014.

[18] Q. T. Tran and C.-Y. Chan. How to ConQueR why-not questions. In *SIGMOD*, pages 15–26, 2010.