Answering Why-not Queries Through Modification-based Explanations

Hiteshi Shah
Advisor: Dr. Carlos Rivero
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
hss7374@cs.rit.edu

***Abstract*—A why-not question is when a user, upon running a query over a database, tries to demand answers as to why certain expected tuple(s) were not included in the result. Our algorithm tries to provide modification-based explanations to such why-not questions by refining the original query in such a way that the new result includes the missing data. The goal here is to implement and evaluate the approach first by modifying the values in the selection predicates and then to try to add selection predicates, as well as consider different schemas for the refined queries that provide some insight to a user's why-not questions at the absence of expected tuples.**

***Index Terms*—why-not queries; conquer algorithm; query analysis**

I. Background

This section provides some background for some of the terms used in this paper. The focus of this project is mainly on select-project-join (SPJ) queries. SPJ queries can take the following form:

SELECT $A_1, A_2, \ldots, A_n$
FROM $R_1, R_2, \ldots, R_n$
WHERE $condition$

Here, R_i refers to a relation in the database and A_i symbolizes an attribute in any of these relations. The condition here is a union of predicates $C_1 \wedge \ldots \wedge C_n$. Each C_i can be either a selection or a join predicate. SPJ queries with aggregation (SPJA) queries consist of SPJ queries along with aggregation operators like SUM, COUNT, AVG, etc. in the SELECT clause and have the optional GROUP BY clause.

II. Introduction

Sometimes when a user runs a query over a database, the query might not return results that the user was expecting. There could be a number of reasons for the missing tuples such as incomplete data if the data was extracted from a different source, or simply tighter constraints in the query that need to be relaxed to be able to see the missing data. The database operations may also be differently interpreted in an application, resulting in discarded data items. Depending on the size of the database, it could be tedious for the user to have to sit through an exhaustive trial-and-error debugging process to try to find the reasons for the missing data. It could also be the case where the user is an application user with no knowledge about the database and no access to the underlying dataset to be able to get the answers for the absent data. Clearly, it would be very helpful for users to seek clarifications on why certain tuples are missing from the result of running a query over a database.

Consider the following SQL query where a user is trying to find a restaurant with a high rating in a particular zip-code: SELECT name FROM restaurants WHERE rating >= 4.0 and zipcode = '14623'. The user could wonder why their favorite pizza place didn't show up in the result. The answer could simply be that their favorite pizzeria falls below a 4.0 rating or is outside the specified zip-code. However, the user may not be able to figure this out without sifting through the entire database or running multiple queries on the system until that particular pizza place shows up in the results.

The current models for answering why-not questions depend on either modifying the database by inserting or updating some data, or identifying the manipulation operators responsible for the missing data. A third method implemented in [1] introduces an algorithm named ConQueR that uses a query refinement approach wherein the algorithm automatically generates a new query by making changes to the original query in such a way that the missing answers show up in the new results while also minimizing the number of irrelevant tuples. Our goal is to implement and evaluate the algorithms described in this paper, first using a relational database and then to try to extend it to document-oriented databases, as well as design a visual tool that can be used in an educational environment that provides some insight to a user's why-not questions at the absence of expected tuples.

III. Related Work

The problem of analyzing why some answers are not included in the output when a query is run has been addressed by various approaches. The main methods for answering why-not questions can be distinguished between instance-based, query-based and modification-based explanations. Instance-based explanations rely on modifying some tuples in the database [2]. The algorithm in [2] focuses on providing explanations for missing answers (or non-answers) to queries that are run

over data extracted from a different source by performing an insert or an update on the data. The result of the query on the modified database includes the original results as well as the missing answers. It is only able to handle SPJ (select-project-join) queries and nothing more complex like queries that aggregate or subqueries.

Artemis [3] extends this algorithm by minimizing the number of insertions performed on the data. It can handle SQL queries that involve selection, projection, join, and union (SPJU), but it cannot handle SPJ queries with aggregation (SPJA).

Query-based explanations of why-not queries rely on identifying the culprit operators in the query plan that are responsible for the discarded data item [4]. The work in [4] can also only handle SPJ queries and not SPJA. K. T. Nicole Bidoit and Melanie Herschel [5] propose a novel algorithm that follows a similar approach of provenance but focuses on SPJA queries.

Top-k queries retrieve k objects that best match user preferences whereas reverse top-k queries try to figure out the user preferences that caused the objects to be in the top-k resultset. Zhian He and Eric Lo [6] and Qing Liu et al. [7] study the problem of answering why-not questions on top-k queries and reverse top-k queries, respectively. Their focus lies on changing the original query and/or k and the preference weights to display the missing answers in the result set.

The algorithm TALOS in [8] implements a query refinement approach where, given the output of some query, the goal is to construct an alternative query such that its result is equivalent to the original result. The paper that we are focusing on [1] is an extension of this work. Similar to Why-Not [4], the ConQueR algorithm [1] tries to find out if the data to generate the absent tuples is available. Then it makes changes to the query in such a way that the output includes the original result as well as the missing tuples. The refined query is as similar as possible to the original query and doesn't output any irrelevant tuples. This work is capable of handling SPJ as well as SPJA queries. Zhian He and Eric Lo [6], Qing Liu et al. [7], Quoc Trung Tran et al. [8], and Quac Trung Tang and Chee-Yong Chan[1] all provide modification-based explanations wherein the original query is refined to provide the missing answers.

Melanie Herschel [9] introduces the novel approach of a hybrid explanation that combines the above exisitng types of explanations. It produces an answer even when no single approach returns the result. It supports SPJA queries as well as queries with one set difference.

The ConQueR algorithm tries to provide a more meaningful explanation to a why-not question by "fixing" the original query rather than merely pointing out the culprit operator(s) [4] or modifying the database [2]. It also makes sure that the refined query is as similar to the original query as possible, unlike [8] which is concerned more with precision in the results.

IV. Overview

This section provides an overview of the implementation and evaluation performed by Quac Trung Tang and Chee-Yong Chan [1]. They have introduced a query refinement algorithm named ConQueR, which is short for Constraint-based Query Refinement. Given a database, a query, and a why-not question, this algorithm automatically constructs explanations to why-not questions by refining the original query. They use two metrics to generate the refined query:

(a) Dissimilarity metric: This metric makes sure that the refined query is as similar to the original query as possible, i.e., the original query has the least modifications done to it. To maximize the similarity between the old and new queries, the algorithm simply modifies some selection predicates in the original query.

(b) Imprecision metric: Ideally, a refined query should only output the original results plus the missing tuples. The imprecision metric is used to minimize the number of irrelevant tuples in the result of the refined query. This is done by adding additional selection predicates to the refined query. The number of selection predicates added should preferably be as low as possible.

Thus, a good refined query has a low dissimilarity metric as well as a low imprecision metric. According to the paper, this can be easily achieved if the original and refined queries share the same schema. If the algorithm cannot find refined queries in the same schema, it generates different schemas by either adding/removing relations or adding/removing join predicates. Then, with the help of a heuristic, it considers these schemas for refined queries in an increasing order of dissimilarity. The refined queries having different schema than the original query tend to have higher values for dissimilarity than the refined queries with the same schema as the original query.

The number of refined queries to be considered can be too large requiring excessive computation. To combat this issue, they have proposed a simpler version of ConQueR named Naive ConQueR or ConQueR$^-$. It uses a heuristic to select a subset of the attributes and makes use of only that subset to generate potential refined queries. They claim that while this method does improve efficiency, the refined queries generated by ConQueR$^-$ tend to have higher dissimilarity values than the queries generated by ConQueR.

To evaluate the efficiency of their algorithms, they conducted experimental studies in two parts. They used two datasets for their experiments: a small synthetic dataset and a real dataset of size 1GB. While they tested both SPJ and SPJA queries on the small dataset, they only tested SPJ queries on the larger dataset with real data. In the first part, they compared the results of ConQueR against the results of ConQueR$^-$ as well as the results of the algorithm TALOS [8] which uses a classification-based approach for constructing refined queries. The algorithms were evaluated on the basis of the dissimilarity and imprecision metrics of the refined queries as well as their runtime performances. In the second part, they did a query-by-query analysis of the results returned by ConQueR and the algorithms in [4] and [2].

While this is one way of evaluating the effectiveness of the algorithm, a more uniform and systematic experimentation would be additionally insightful. We could study the perfor-

mance of the algorithm by analyzing the impact of each of its parameters on the performance by changing the value of one parameter at a time while keeping the others fixed to their default values. We could also try to gauge how meaningful and helpful the answers to the why-not questions are to the end user and see if having these explanations enhances user understanding over having no explanantions. In the end, we would like the run the ConQueR algorithm as well as our algorithm on the same dataset and compare the results.

V. Implementation

Since we are interested in making the process of finding refined queries as automatic as possible, we have automated the task of finding the why-not questions, instead of having the user provide the questions. Given a query, we randomly modify it in such a way that its resultset misses tuples from the original resultset. The modification to the original query can be as simple as changing the values in the selection predicates or using completely different joins. This process is also automatic. The user must provide rules for changing the attribute values in the following format: "*table-name.attribute-name -> value$_1$ value$_2$ value$_3$ ...; operator*". Upon feeding the original query as well as the modified query to the algorithm, the algorithm finds the missing tuples, picks a random percent of them, and considers them as the "why-not questions". The steps for finding the why-not questions are pretty straightforward:

Algorithm 1 Getting Why-Not Questions

Input: Original Query, *OQ*
Input: Modified Query, *Q*
Input: Random value between 0 and 1, *percentage*
Data: Dataset, *D*
Output: Set of Why-Not Questions, *S*

1: *OQ(D)* = results of query *OQ* on dataset *D*
2: *Q(D)* = results of query *Q* on dataset *D*
3: *S* = *percentage* * (set of why-not questions, i.e., tuples in *OQ(D)* that are not in *Q(D)*)

The program then outputs refined queries that answer the why-not questions. For ease of computation in our running example, we have assumed that the input queries only consist of the $\leq$ sign, the AND operator, and numerical attributes. The algorithms can be easily tweaked to work for the other operators. The format of why-not questions is: *S* = {{<*attribute-name$_1$*>, <*attribute-value*>, <*attribute-name$_2$*>, <*attribute-value*>, ...}, {<*attribute-name$_1$*>, <*attribute-value*>, <*attribute-name$_2$*>, <*attribute-value*>, ...}, ...}.

The implementation for obtaining refined queries is divided into three sections:

A. Modifying Selection Predicates: This section tries to include the missing tuples in the results by changing the values in the selection predicates.
B. Adding Selection Predicates: If the refined query outputted by the previous section results has any extra tuples, i.e., tuples that don't belong to the results of the modified query or the why-not questions, then this section focuses on adding more selection predicates to the refined query in order to get rid of the irrelevant tuples.
C. Adding/Removing Joins: It is possible that the tables in the modified query do not account for some or any of the why-not questions. In such cases, the algorithm tries to add/remove joins in the query until the new query result includes the why-not questions as well as the result of the modified query. It then repeats section B to reduce the number of irrelevant tuples.

Running Example: In this paper, we use the IMDb dataset as our running example which consists of three main tables:

- *Member* (*id*, *name*, *birthYear*, *deathYear*)
- *Movie_Actor* (*actor*, *movie*)
  - *actor* FK *Member*(*id*)
  - *movie* FK *Movie*(*id*)
- *Movie_Writer* (*writer*, *movie*)
  - *writer* FK *Member*(*id*)
  - *movie* FK *Movie*(*id*)

Since the dataset is too large for quick computations, for this example, we have randomly selected eight rows from the *Movie_Actor* table and six rows from the *Movie_Writer* table. Our datasets of the selected tuples from the *Movie_Actor* and *Movie_Writer* tables and the corresponding tuples in the *Member* table can be seen in Tables 1, 2 and 3, respectively. In order to save some writing space, we shall henceforth be refering to the attributes "*birthYear*" as "*bY*" and "*deathYear*" as "*dY*".

actor	*movie*
65159	152216
1234	48750
522884	1109104
361127	6259478
808485	1382115
139632	5811388
289972	2246373
2042	397241

Table 1: *Movie_Actor*

writer	*movie*
2042	397241
463172	530495
1175	673608
725034	811123
186371	1419199
1760975	1747397

Table 2: *Movie_Writer*

A. Modifying Selection Predicates

Example 1: Consider *OQ* = "SELECT *name* FROM *Member* m, *Movie_Actor* a WHERE m.*id* = a.*actor* AND *bY* <= 1950 AND *dY* <= 2000". The resultset of this query on our

id	*name*	*birthYear*	*deathYear*
65159	Jackson Beck	1912	2004
1234	John Forsythe	1918	2010
522884	Arthur Lowe	1915	1982
361127	William Hansen	1911	1975
808485	Howard K. Smith	1914	2002
139632	Carole Carr	1928	1997
289972	Sergio Franchi	1926	1990
2042	Charles Dickens	1812	1870
463172	David Kohan	1964	NULL
1175	Blake Edwards	1922	2010
725034	Jason Richman	NULL	NULL
186371	John Crane	NULL	NULL
1760975	Claudio Descalzi	NULL	NULL

Table 3: *Member*

dataset D is given in Table 4. Now, if OQ is modified to look like Q = "SELECT *name* FROM *Member* m, *Movie_Actor* a WHERE m.*id* = a.*actor* AND $bY <= 1950$ AND $dY <= 1985$", we get the resultset in Table 5.

Algorithm 2 Modifying Selection Predicates

Input: Modified Query, Q
Input: Set of Why-Not Questions, S
Data: Dataset, D
Output: Set of Refined Queries, Q'

1: $Q^*_\phi = Q$ on replacing the column names in SELECT with '*' and removing all selection predicates
2: $Q^* = Q^*_\phi$ with the selection predicates from Q
3: $Q^*_\phi(D)$ = results of query Q^*_ϕ on dataset D
4: $Q^*(D)$ = results of query Q^* on dataset D
5: v^{max} = maximum of the values for selection predicate attributes in the tuples in $Q^*(D)$
6: $SL = \emptyset$
7: **for** $s \in S$ **do**
8: M = set of tuple numbers in $Q^*_\phi(D)$ that match s
9: sl = set of tuple numbers in M such that the values for selection predicate attributes in a tuple $\leq$ values of every other tuple in M and that at least one of these inequalities is strict
10: $SL := SL \cup \{sl\}$
11: M' = set of skyline tuples such that it has one tuple number from each sl in SL
12: $Q' = \emptyset$
13: **for** $m' \in M'$ **do**
14: v^{max} = maximum of the values for selection predicate attributes in the tuples in m' and the previous values of v^{max}
15: q' = Q on replacing the values of the attributes in the selection predicates with the values from v^{max}
16: $Q' := Q \cup \{q'\}$

name
Arthur Lowe
William Hansen
Carole Carr
Sergio Franchi
Charles Dickens

Table 4: *OQ(D)*

name
Arthur Lowe
William Hansen
Charles Dickens

Table 5: *Q(D)*

Thus, the set of why-not questions S obtained from Algorithm 1 will consist of "Carole Carr" and "Sergio Franchi". In this example, the random *percentage* = 1, so S = {{"*name*", "Carole Carr"}, {"*name*", "Sergio Franchi"}}.

Now, Q^*_ϕ = "SELECT * FROM *Member* m, *Movie_Actor* a WHERE m.*id* = a.*actor*", the resulset of which is given in Table 6. Simiarly, adding the selection predicates from Q to Q^*_ϕ, we get Q^* = "SELECT * FROM *Member* m, *Movie_Actor* a WHERE m.*id* = a.*actor* AND $bY <= 1950$ AND $dY <= 1985$". Its resultset $Q^*(D)$ is shown in Table 7. We know the selection predicate attributes are "*bY*" and "*dY*". The maximum value for bY is 1915 and the maximum value for dY is 1982 in $Q^*(D)$. So, v^{max} = {"1915", "1982"}.

Now, considering the first tuple in S, we get s = {"*name*", "Carole Carr"}. Here, $M = \{t_6\}$ and since there is only one matching tuple, $sl = \{t_6\}$. Similarly, for the second tuple in S, we get s = {"*name*", "Sergio Franchi"}. Here, $M = \{t_7\}$ and since there is only one matching tuple, $sl = \{t_7\}$. Thus, $SL = \{\{t_6\}, \{t_7\}\}$. $M' = \{\{t_6, t_7\}\}$ since it takes one tuple number from each row in SL.

	id	*name*	*bY*	*dY*	*movie*
t_1	65159	Jackson Beck	1912	2004	152216
t_2	1234	John Forsythe	1918	2010	48750
t_3	522884	Arthur Lowe	1915	1982	1109104
t_4	361127	William Hansen	1911	1975	6259478
t_5	808485	Howard K. Smith	1914	2002	1382115
t_6	139632	Carole Carr	1928	1997	5811388
t_7	289972	Sergio Franchi	1926	1990	2246373
t_8	2042	Charles Dickens	1812	1870	397241

Table 6: $Q^*_\phi(D)$ where D = {*Member*, *Movie_Actor*}

	id	*name*	*bY*	*dY*	*movie*
t_3	522884	Arthur Lowe	1915	1982	1109104
t_4	361127	William Hansen	1911	1975	6259478
t_8	2042	Charles Dickens	1812	1870	397241

Table 7: $Q^*(D)$

Considering each set in M', we get $m' = \{t_6, t_7\}$. Among t_6 and t_7, the maximum value for bY is 1928 and the maximum value for dY is 1997. Comparing these maximum values with the current values in v^{max}, we get maximum value as 1928 for bY and 1997 for dY, i.e., v^{max} = {"1928", "1997"}. Replacing the values for the selection predicate attributes in Q using the values in v^{max}, we get q' = "SELECT *name* FROM *Member* m, *Movie_Actor* a WHERE m.*id* = a.*actor* AND $bY <= 1928$ AND $dY <= 1997$". And since there is only one set in M', Q' =

{"SELECT *name* FROM *Member* m, *Movie_Actor* a WHERE m.*id* = a.*actor* AND *bY* <= 1928 AND *dY* <= 1997"}.

The resultset for q' is shown in Table 8. As we can see, these results match the resulset of the original query.

name
Arthur Lowe
William Hansen
Carole Carr
Sergio Franchi
Charles Dickens

Table 8: Example 1: *q'(D)*

B. Adding Selection Predicates

This algorithm takes as input the outputs from Algorithm 2.

Example 2: Consider the inputs *OQ* = "SELECT *name* FROM *Member* m, *Movie_Actor* a WHERE m.*id* = a.*actor* AND *bY* <= 1927", *Q* = "SELECT *name* FROM *Member* m, *Movie_Actor* a WHERE m.*id* = a.*actor* AND *bY* <= 1915" and *percentage* = 0.5 being passed to Algorithm 1. So, here, *S* = {{"*name*", "Sergio Franchi"}} and Algorithm 2 gives the output *Q'* = {"SELECT *name* FROM *Member* m, *Movie_Actor* a WHERE m.*id* = a.*actor* AND *bY* <= 1926"}. The resultset for q' is shown in Table 9. As we can see, these results contain an extra tuple "John Forsythe" that doesn't belong to *Q(D)* or *S*. Thus, q' will be given as input to Algorithm 3, along with *S*, *Q(D)*, $q'^{*}_{\phi}(D)$ (Table 6), and *isize* = 6. The ideal resultset size, *isize*, is calculated as the sum of the sizes of *Q(D)* and *S*, since ideally, we want the refined query output to include only *Q(D)* and *S*.

name
Jackson Beck
John Forsythe
Arthur Lowe
William Hansen
Carole Carr
Sergio Franchi
Charles Dickens

Table 9: Example 2: *q'(D)*

Algorithm 3 will first run q' over *D*, which is already shown in Table 9. Then, it gets *size* = 7 and *ATTR* = {"*dY*"} (*ATTR* ignores key attributes as well as the attributes in the SELECT-clause). Here, $M = \{t_1, t_3, t_4, t_5, t_6, t_7, t_8\}$ and since there is only one attribute in *ATTR*, *attr* = "*dY*". Now, v^{max} = 2004 for *attr* = "*dY*", so q'' = "SELECT *name* FROM *Member* m, *Movie_Actor* a WHERE m.*id* = a.*actor* AND *bY* <= 1926 AND *dY* <= 2004". The resultset for q'' is shown in Table 10.

The algorithm first checks if q'' is valid, i.e., it contains both *S* as well *Q(D)*. Since the new query is valid, it calculates *nsize* = 6 which is >= *isize* (6) and < *size* (7), and thus the refined query q' is set to q'' and returned.

Algorithm 3 Adding Selection Predicates

Input: Refined Query, q'
Input: Set of Why-Not Questions, *S*
Input: Resultset of *Q* on dataset *D*, *Q(D)*
Input: Resultset of q'^{*}_{ϕ} on dataset *D*, $q'^{*}_{\phi}(D)$
Input: Ideal Resultset Size, *isize*
Data: Dataset, *D*
Output: Refined Query, q'

1: $q'(D)$ = results of query q' on dataset *D*
2: *size* = size of $q'(D)$
3: *ATTR* = attributes in $q'^{*}_{\phi}(D)$ - selection attributes
4: $M = \emptyset$
5: **for** $s \in S$ **do**
6: $M := M \cup \{$tuple number of *s* in $q'^{*}_{\phi}(D)\}$
7: **for** *tuple* $\in Q(D)$ **do**
8: $M := M \cup \{$tuple number of *tuple* in $q'^{*}_{\phi}(D)\}$
9: **for** *attr* $\in$ *ATTR* **do**
10: v^{max} = maximum of the values in *attr* for the tuples in *M*
11: q'' = q' on adding the new attribute *attr* with value v^{max}
12: $q''(D)$ = results of query q'' on dataset *D*
13: **if** q'' is valid **then**
14: *nsize* = size of $q''(D)$
15: **if** *nsize* >= *isize* & *nsize* < *size* **then**
16: $q' = q''$

name
Jackson Beck
Arthur Lowe
William Hansen
Carole Carr
Sergio Franchi
Charles Dickens

Table 10: $q''(D)$

C. Adding/Removing Joins

Since the process of modifying the original query is automatic, it is possible that the tables in the modified query do not match the tables in the original query.

Example 3: Consider the inputs *OQ* = "SELECT *name* FROM *Member* m, *Movie_Writer* w WHERE m.*id* = w.*writer* AND *bY* <= 1927", *Q* = "SELECT *name* FROM *Member* m, *Movie_Actor* a WHERE m.*id* = a.*actor* AND *bY* <= 1900" and *percentage* = 1 being passed to Algorithm 1. As we can see, query *OQ* is run over tables *Member* and *Movie_Writer* whereas query *Q* is run over tables *Member* and *Movie_Actor*. Let D_1 = {*Member*, *Movie_Writer*} and D_2 = {*Member*, *Movie_Actor*}. The resultsets $OQ(D_1)$ and $Q(D_2)$ are shown in Tables 11 and 12 respectively.

name
Charles Dickens
Blake Edwards

Table 11: $OQ(D_1)$

name
Charles Dickens

Table 12: $Q(D_2)$

Here, S = {{"*name*", "Blake Edwards"}}. Since Algorithm 2 works on the dataset of Q, we first need to make sure that $Q^*_\phi(D_2)$ contains "Blake Edwards". But as we can see from Table 6, this value does not belong to the joined table of D_2 and thus Algorithm 2 would fail.

In such cases, we need to first find a join that contains both $Q(D_2)$ as well as S = {{"*name*", "Blake Edwards"}} by considering all the possible joins in the database schema. In this example, we find that query q' = "SELECT *name* FROM *Member* m, *Movie_Writer* w WHERE m.*id* = w.*writer*" fits the bill. The join for the tables in q' is shown in Table 13.

	id	*name*	*bY*	*dY*	*movie*
t_1	2042	Charles Dickens	1812	1870	397241
t_2	463172	David Kohan	1964	NULL	530495
t_3	1175	Blake Edwards	1922	2010	673608
t_4	725034	Jason Richman	NULL	NULL	811123
t_5	186371	John Crane	NULL	NULL	1419199
t_6	1760975	Claudio Descalzi	NULL	NULL	1747397

Table 13: $Q^*_\phi(D)$ where D = {*Member*, *Movie_Writer*}

As we can see, this query gives out some extra tuples like "David Kohan, "Jason Richman", "John Crane" and "Claudio Descalzi". So to reduce them, we pass the query q' to Algorithm 3, along with $S = S \cup Q(D)$, $q'^*_\phi(D_1)$ (Table 13), $q'^*(D_1) = \emptyset$ and $isize$ = 2, which then sets q' to "SELECT *name* FROM *Member* m, *Movie_Writer* w WHERE m.*id* = w.*writer* AND *bY* <= 1922", the resultset for which is show in Table 14.

name
Charles Dickens
Blake Edwards

Table 14: Example 3: $q'(D)$

However, in some cases, it may be possible that no join exists that accounts for both the why-not questions and the results of the modified query. In such cases, we simply pass a query the only contains the tables corresponding to the attributes in the SELECT-clause along with S to Algorithm 2 and combine its resulting refined query with the modified query using the SQL keyword "UNION".

VI. Evaluation

In this section, we evaluate the efficiency and effectiveness of our implementation. The program in this paper was coded in Java, and MySQL Server 8.0.13 was used as our relational database system. The experiments were conducted on a Windows 10 PC with a 8GB RAM and a 250GB SSD.

For our experiments, we're making use of the *Movie* table with 1000 tuples, the *Movie_Actor* table with 927 tuples, the *Movie_Writer* table with 557 tuples, the *Movie_Producer* table with 477 tuples, and the *Movie_Director* table with 798 tuples from the IMDB dataset.

The evaluation is done in two parts. We first check if a refined query given out by the program is valid. The second part of the evaluation involves calculating the total number of true positives, false positives and false negatives in the results of the refined query w.r.t the results of the original query. Let R be the resultset of the original query, R' the resultset of the modified query, and R'' the resultset of the refined query. Let the why-not questions be represented by S.

1) Validation: $R'' = R' \cup S$
2) Accuracy: For a tuple t
   a) True Positive: $t \in R$ & $t \in R''$
   b) False Positive: $t \in R''$ & $t \notin R$
   c) False Negative: $t \in R$ & $t \notin R''$

The refined queries in Tables 15 and 16 obtained from our experiments are all valid. The first and second columns in the tables refer to the original and modified queries, respectively. The third column is the percentage (%) of why-not questions that we are keeping. The fourth column is the number of why-not questions left and passed to the algorithm. The fifth column is a valid refined query given out by the program for the provided modified query and why-not questions. The sixth column is the amount of time (in seconds) it took for the program to finish executing. And the final three columns are the number of true positives (TP), false positives (FP), and false negatives (FN) w.r.t the results of the original and refined queries. The reason for calculating these values is to see how close the results of the refined query are to the those of the original query.

We wanted to see the how the program fares when it is provided with queries that were modified depth-wise as well as breadth-wise. The experiments in Table 15 are done using only one user-provided rule for new attribute values: "movie.startYear -> 1920 1950 1980 2010; <=". In the first row, we see that the modified query has the same joins as the original query but the value of its selection attribute is changed. We also see that when over 75% of the why-not questions are kept, the refined query is the same as the original query. However, when only 50% or 25% of the why-not questions are kept, the program adds some selection predicates to the previous refined query. This affects the number of TP, FP, and FN, as seen in the table. We can also see that the program gives out refined queries for this modified query much faster than the modified query in the second row.

In the second row, we see that the modified query has different joins and attribute values than the original query. This results in the "UNION" keyword in the refined queries. Similar to the first row, when over 75% of the why-not questions are kept, the refined query returns the original query and combines it with the modified query. When only 50% or 25% of the why-not questions are kept, the program adds some selection predicates to the original refined query before combining it with the modified query. The number of TP, FP, and FN are similarly affected.

The experiments in Table 16 are done using two user-provided rules for new attribute values: "movie.startYear -> 1920 1950 1980 2010; <=" and "movie.numVotes -> 10 250 1000 2500; <=". Similar to the other set of experiments, the first row has a modified query with the same joins as the original query and different selection attribute values. We see that when all of the why-not questions are kept, the refined query has the same selection attributes as the original query but with different value. However, when why-not questions are removed, the program adds some selection predicates to the refined query along with changing the values. There isn't much of a time difference in the program execution between the two rows.

Once again, the second row has a modified query with different joins and attribute values than the original query which results in the "UNION" keyword in the refined queries. It produces similar results as discussed for the previous table. The number of true positives decreases as the number of why-not questions decreases while the number of false positives/negatives increases.

VII. Comparison of Models

In this section, we evaluate the results of the algorithms implemented in this paper with the results of ConQueR [1]. ConQueR [1] tests its algorithm using the Basketball dataset (from http://www.basketballreference.com) and the TPC-H benchmark. Since the TPC-H dataset is more accessible, we decided to replicate the queries and why-not questions used for this dataset in [1] and compare the results. The comparison can be seen in Table 17.

Here, 'd' refers to the dissimilarity metric and 'i' refers to the imprecision metric. The dissimilarity metric is calculated by incrementing it by 1 when a selection attribute value is changed, by incrementing it by 3 when a selection attribute

Original Query	Modified Query	%	# why-not	Refined Query	time	TP	FP	FN
SELECT title FROM movie m, movie_actor a WHERE m.id = a.movie AND startYear <= 1960	SELECT title FROM moviet, movie_actor WHERE movie.id = movie_actor.movie AND startYear <= 1920	100	92	SELECT title FROM movie, movie_actor WHERE movie.id = movie_actor.movie AND startYear <= 1960	1.6s	94	0	0
		75	69	SELECT title FROM movie, movie_actor WHERE movie.id = movie_actor.movie AND startYear <= 1960	1.6s	94	0	0
		50	46	SELECT title FROM movie, movie_actor WHERE movie.id = movie_actor.movie AND startYear <= 1960 AND runtime <= 130	1.6s	93	1	0
		25	23	SELECT title FROM movie, movie_actor WHERE movie.id = movie_actor.movie AND startYear <= 1960 AND runtime <= 106 AND numVotes <= 1346	1.7s	84	10	0
SELECT title FROM movie m, movie_actor a WHERE m.id = a.movie AND startYear <= 1960	SELECT title FROM movie, movie_director WHERE movie.id = movie_director.movie AND startYear <= 1920	100	92	SELECT title FROM movie, movie_director WHERE movie.id = movie_director.movie AND startYear <= 1920 UNION SELECT title FROM movie WHERE startYear <= 1960	2.4s	94	11	0
		75	69	SELECT title FROM movie, movie_director WHERE movie.id = movie_director.movie AND startYear <= 1920 UNION SELECT title FROM movie WHERE startYear <= 1960	2.5s	94	11	0
		50	46	SELECT title FROM movie, movie_director WHERE movie.id = movie_director.movie AND startYear <= 1920 UNION SELECT title FROM movie WHERE startYear <= 1960 AND numVotes <= 3567	2.4s	92	11	2
		25	23	SELECT title FROM movie, movie_director WHERE movie.id = movie_director.movie AND startYear <= 1920 UNION SELECT title FROM movie WHERE startYear <= 1956 AND runtime <= 111 AND numVotes <= 3117	2.3s	88	11	6

Table 15: Evaluation where only one rule for "startYear" is passed by the user for query modification

Original Query	Modified Query	%	# why-not	Refined Query	time	TP	FP	FN
SELECT title FROM movie m, movie_actor a WHERE m.id = a.movie AND startYear <= 1960 AND numVotes <= 500	SELECT title FROM movie, movie_actor WHERE movie.id = movie_actor.movie AND startYear <= 1920 AND numVotes <= 250	100	81	SELECT title FROM movie, movie_actor WHERE movie.id = movie_actor.movie AND startYear <= 1960 AND numVotes <= 461	2.2s	83	0	0
		75	61	SELECT title FROM movie, movie_actor WHERE movie.id = movie_actor.movie AND startYear <= 1960 AND numVotes <= 461 AND runtime <= 130	2.2s	82	0	1
		50	41	SELECT title FROM movie, movie_actor WHERE movie.id = movie_actor.movie AND startYear <= 1960 AND numVotes <= 461 AND runtime <= 130	2.5s	75	0	8
		25	20	SELECT title FROM movie, movie_actor WHERE movie.id = movie_actor.movie AND startYear <= 1960 AND numVotes <= 95 AND runtime <= 115	2.6s	64	0	19
SELECT title FROM movie m, movie_actor a WHERE m.id = a.movie AND startYear <= 1960 AND numVotes <= 500	SELECT title FROM movie, movie_director WHERE movie.id = movie_director.movie AND startYear <= 1920 AND numVotes <= 250	100	81	SELECT title FROM movie, movie_director WHERE movie.id = movie_director.movie AND startYear <= 1920 AND numVotes <= 250 UNION SELECT title FROM movie WHERE startYear <= 1960 AND numVotes <= 461	2.4s	83	9	0
		75	61	SELECT title FROM movie, movie_director WHERE movie.id = movie_director.movie AND startYear <= 1920 AND numVotes <= 250 UNION SELECT title FROM movie WHERE startYear <= 1960 AND numVotes <= 461	2.5s	83	9	0
		50	41	SELECT title FROM movie, movie_director WHERE movie.id = movie_director.movie AND startYear <= 1920 AND numVotes <= 250 UNION SELECT title FROM movie WHERE startYear <= 1960 AND runtime <= 109 AND numVotes <= 461	2.4s	79	9	4
		25	20	SELECT title FROM movie, movie_director WHERE movie.id = movie_director.movie AND startYear <= 1920 AND numVotes <= 250 UNION SELECT title FROM movie WHERE startYear <= 1959 AND runtime <= 106 AND numVotes <= 387	2.3s	70	9	13

Table 16: Evaluation where two rules, "startYear" and "numVotes", are passed by the user for query modification

is added, by incrementing it by 5 when a join predicate is added/removed and by incrementing it by 7 when a relation is added/removed. The imprecision metric is calculated by counting the number of irrelevant tuples in the results of the refined query w.r.t the results of the modified query.

So, as we can see from Table 17, the results of our algorithm match the results of ConQueR for the two queries listed. There is a third query tested for TPC-H in [1]: "SELECT part.p_name FROM part, partsupp WHERE part.p_partkey = partsupp.ps_partkey AND p_retailprice > 800 AND ps_supplycost > 990". However, we were unable to find the exact why-not question that they used for this query ("beige steel") in our version of the TPC-H dataset. We did find that "beige steel" was a substring of several results in the join of the tables "part" and "partsupp". So using each of those as the why-not questions one-by-one for the above query, we found the results that matched the ConQueR results closest and furthest. [1] gave d = 2 and i = 762 for this query. The closest we got was with the why-not question "beige steel aquamarine lavender blanched" that gave d = 5 and i = 1554, and the furthest was the why-not question "burlywood midnight slate beige steel" with d = 5 and i = 156248.

VIII. Conclusion

The algorithm implemented in this paper automatically explains questions by making changes to the query. This

Query	Why-Not Questions	ConQueR		Our Algorithms	
		d	i	d	i
SELECT supplier.s_name FROM supplier, nation WHERE supplier.s_nationkey = nation.n_nationkey AND s_acctbal $>$ 1000 AND n_regionkey $<$ 5	Supplier1336, Supplier9819	2	2197	2	2197
SELECT customer.c_name FROM customer, nation WHERE customer.c_nationkey = nation.n_nationkey AND c_acctbal $>$ 9800	Customer100, Customer197, Customer219, Customer468, Customer518, Customer780, Customer987, Customer1042, Customer1370, Customer1573	1	1296	1	1296

Table 17: Comparison of models

approach always gives an answer as long as the why-not tuples are accesible, and the refined queries produced by it are close in similarity to the original query. The one drawback that was noticed while evaluating the algorithm was that the program takes very long to execute if there are a large number of tuples in the database that match the why-not questions. For future work, we would like to extend this algorithm to work for SPJA queries and also take into account data types other than the ones mentioned in this paper.

REFERENCES

[1] C.-Y. C. Quac Trung Tang, "How to conquer why-not questions," *SIGMOD*, pp. 15–26, 2010.

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

[3] W.-C. T. Melanie Herschel, Mauricio A. Hernández, "Artemis: a system for analyzing missing answers," *VLDB*, pp. 1550–1553, 2009.

[4] H. V. J. Adriane Chapman, "Why not?" *SIGMOD*, pp. 523–534, 2009.

[5] K. T. Nicole Bidoit, Melanie Herschel, "Query-based why-not provenance with nedexplain," *EDBT*, p. 145–156, 2014.

[6] E. L. Zhian He, "Answering why-not questions on top-k queries," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1300–1315, 2014.

[7] G. C.-B. Z. L. Z. Qing Liu, Yunjun Gao, "Answering why-not and why questions on reverse top-k queries," *VLDB*, pp. 867–892, 2016.

[8] S. P. Quoc Trung Tran, Chee-Yong Chan, "Query by output," *SIGMOD*, pp. 535–548, 2009.

[9] M. Herschel, "Wondering why data are missing from query results?: ask conseil why-not," *CIKM*, pp. 2213–2218, 2013.