Query-based Why-not Explanations for Nested Data

Ralf Diestelkämper
IPVS - University of Stuttgart
ralf.diestelkaemper@ipvs.uni-stuttgart.de

Boris Glavic
Illinois Institute of Technology
bglavic@iit.edu

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

Seokki Lee
Illinois Institute of Technology
slee195@hawk.iit.edu

Abstract

We present the first query-based approach for explaining missing answers to queries over nested relational data which is a common data format used by big data systems such as Apache Spark. Our main contributions are a novel way to define query-based why-not provenance based on repairs to queries and presenting an implementation and preliminary experiments for answering such queries in Spark.

Keywords Why-not provenance, nested data

1 Introduction

The need to explain missing answers is prevalent in many applications including debugging complex analytical queries. Such complex analytics are often implemented using data-intensive scalable computing (DISC) systems such as Spark which employ nested data models. Hence, there is a need for missing answer techniques for nested data models and implementations of such techniques in DISC systems.

Missing answers approaches typically fall into one of two categories: *instance-based* approaches justify the absence of an answer based on missing input data; and *query-based* approaches explain the missing answer by determining which parts of the query are responsible for the failure to derive the result. The contributions of this work towards explanation for query-based missing-answers are twofold: (i) We determine which operators are responsible for a missing answer based on a novel notion of responsibility rooted in query repairs. This is necessary as the notion of *picky operators* that was introduced in the seminal work by Chapman [5] and was used as the basis of most follow-up work on query-based why-not provenance is not suited well for nested data as we illustrate below; (ii) we present a practical solution for explaining missing answers in this context, a prototype implementation using provenance tracking for Spark.

Example 1.1. Consider the nested relation shown in Figure 1a, which records taxi rides. Each row corresponds to one car, storing the `driver` in a nested tuple and `rides` in a nested bag. The `rides` are grouped by their origin (`from`) and destination (`to`). For each origin and destination, the relation stores the `start` and `end` times of all trips in a nested bag `times`. The Spark program in Figure 1c groups trips after 10 am by their origin and destination, and computes a nested bag of `drivers` for each trip. The result is shown in Figure 1b. Looking at the 2^{nd} result tuple, a user may wonder why there is no result with the same `from` and `to` values where (i) Jane is a driver or (ii) John appears only once instead of twice. Our query-based why-not approach identifies which operators of the Spark program cause the unexpected result. Here, the filter operator or the second flatten operator are too restrictive for (i). The filter condition can be changed to 9 am or the second flatten operator should unnest the *end* times instead of the *start* times. For (ii) the filter operator it not restrictive enough. Why-not techniques such as [5] identify *picky operators*, i.e., operators whose inputs contain data items that are "successors" of (are derived from) an input which is "unpicked" (is compatible with the missing answer), but whose output does not contain data items derived from such successors. The intuition is that these operators are responsible for removing data that could potentially have produced the missing answer. However, such an approach

driver		car	rides			
id	name		from	to	times	
					start	end
1	John	SX-123-2B	1st St.	10th St.	09 : 45	10 : 16
			Main St.	Market St.	start	end
					10 : 29	10 : 42
					12 : 12	12 : 23
id	name		from	to	times	
					start	end
2	Jane	VW-678-5X	1st St.	10th St.	17 : 20	17 : 36
			Main St.	Market St.	start	end
					09 : 52	10 : 11

(a) Input relation `taxiRides`

from	to	driver
		name
1st St.	10th St.	Jane
		name
Main St.	Market St.	John
		John

(b) Output

```
val drs = spark.read.json("taxiRides.json")
val fl1 = drs.withColumn("ride",explode($"rides"))
val fl2 = fl1.withColumn("time",explode($"ride.times.start"))
val fi = fl2.filter($"time" > "10:00")
val rt = fi.select($"ride.from", $"ride.to", $"driver.name")
val rs = rt.groupBy($"from", $"to").agg(collect_list($"name"))
```

(c) Processing pipeline

Figure 1. Given relation `taxiRides` (a), program (c) computes a list of drivers (b) for each route (`from` A `to` B) .

TaPP 2019, June 3, 2019, Philadelphia, PA

would not identify the filter and flatten operators in our example as picky wrt. the question (i), since a successor of the 2^{nd} input tuple which is compatible with (i) exists in the output (the 2^{nd} result tuple).

2 Related Work

The work presented in this paper mainly builds on prior research in three areas. We refer readers to [9] for a survey.

Query-based why-not provenance. Solutions that explain missing results based on the query or data transformation either identify query operators (query-based, e.g., [3–5]) responsible for the disappearance of relevant data or "repair" the query by changing selected operators such that the missing result is returned. The later have also been referred to as query refinement [13] and query relaxations [12]. All these approaches target flat relational data and, with the exception of [2, 5] which target workflows, support queries limited to (at best) relational algebra plus grouping and aggregation. These approaches cannot be easily extended to a richer set of operators and nested data.

Provenance capture in DISC systems. While research on why-not provenance has mostly focused on relational queries, provenance capture for DISC systems has been studied [1, 10]. Given that we target query-based why-not explanations for nested data in the context of such systems, we choose to extend one of these systems.

Provenance for nested data. Given that query-based why-not explanations are typically defined and computed based on the provenance of existing results, our work is also related to work on provenance models for nested data [1, 6].

3 Preliminaries and Notation

To define why-not questions and explanations for nested data, we first formalize the nested relational model and a bag version of relational algebra (NRAB) inspired by [7]. Similar to [1], this allows us to express a large variety of practical queries on big data analytics platforms.

Definition 3.1 (Nested Relation)**.** Let $\mathbb{L}$ be an infinite set of names. A *nested instance I* is an element of a type τ conforming to the grammar shown below where each $A_i \in \mathbb{L}$. A *nested relation* is a nested instance of some $\mathcal{R}$-type.

$$\mathcal{P} := \text{INT} \mid \text{STR} \mid \text{BOOL} \mid \ldots \quad \mathcal{R} := \{\{\mathcal{T}\}\}$$
$$\mathcal{T} := \langle A_1 : \mathcal{A}, \ldots, A_n : \mathcal{A}\rangle \quad \mathcal{A} := \mathcal{P} \mid \mathcal{T} \mid \mathcal{R}$$

Data conforming to this model is manipulated through NRAB using the operators shown in Table 2. Table 1 provides an overview of the notation used here. Due to space constraints, we cannot discuss all details. We also omit the definition of operators which behave as in flat relational algebra: table access, projection, renaming, selection, join, union, and deduplication. Given that our data model allows a nested element to either be a tuple or a bag of homogeneous tuples (i.e., a nested relation), we provide two types of nesting and flatten operators – one for nested tuples and one for nested relations. Aggregation applies to each input tuple individually, aggregating all values from a relation nested in a specified attribute.

Notation	Description
$t.A$	Projection of tuple t on an attribute of set of attributes A
$\tau_{R.A}$	The tuple type of a projection of relation R on attribute(s) A, e.g., if $type(R) = \{\{\langle A_1 : \text{INT}, A_2 : \text{INT}\rangle\}\}$ then $\tau_{R.A_1} = \langle A_1 : \text{INT}\rangle$
$\text{SCH}(R)$	The set of attributes of relation R
$\circ$	Concatenation of tuples and tuple types
$t^n \in R$	Tuple t appears in relation R with multiplicity n. For convenience, $t^0 \in R$ indicates that t does not appear in R
$[\![Q]\!]_D, [\![Q]\!]$	Evaluating query Q over a database D, possibly omitting D
$Q(D)$	The result of evaluating a query Q over input D

Table 1. Notation

Example 3.2. Employing our algebra, the sample program of Figure 1c can be expressed as follows:

$$A_{name \to driver}(\pi_{from,to,name}(\sigma_{time>10:00}($$
$$F_{driver}(F_{ride.times.start \to time}(F_{rides \to ride}(\texttt{taxiRides})))))$$

We first denormalize the whole relation by flattening the rides, times, and driver elements. We then filter records based on the trip duration and project onto `from`, `to`, and `name`. Finally, we nest all drivers for a trip in an element `driver`.

4 Why-Not Provenance for Nested Data

We now introduce why-not questions, (minimal) successful re-parameterizations, and explanations.

4.1 Why-Not Questions

In our framework, a why-not question consists of a query, a database instance, and a missing tuple with constants and placeholders. The purpose of placeholders is to give the user the flexibility to leave some parts of the missing answer of interest unspecified, e.g., in our running example the user may ask why are there no rides in the result that start from 10th St. (leaving the destination and set of drivers unspecified).

Definition 4.1 (Why-not question)**.** A why-not question Φ is a triple (Q, D, t) where Q is a query, D is a nested relation, and t is a tuple that is of same type as the result of query Q. Each element in t is either (i) a constant value of the element's type, (ii) a placeholder ? that denotes some value of the type of the element, or (iii) a placeholder $*$ that represents any number of tuples (possibly none) in nested collections.

Example 4.2. The why-not questions (i) and (ii) from Example 1.1 can be expressed using the tuples shown below.

t_1=⟨from:Main St., to:Market St.,driver:{{⟨name:Jane⟩,∗}}⟩
t_2=⟨from:Main St., to:Market St.,driver:{{⟨name:John⟩}}⟩

If instead we are interested in why there are no trips starting from 10th street in the result, we can use placeholders for attributes `to` and `driver`: t_3 = ⟨from:10th St., to:?, driver:?⟩

Note that for finite domains, the problem of answering why-not questions with placeholders can be reduced to answering a set of questions where the tuple is fully specified using constants.[1] However, this reduction may come at the

[1]Even though there are infinitely many instances of a nested relation since we employ bag semantics, the number of instances that can be produced by repairs of a query are finite since the input database is fixed.

Operator	Semantics	Output type $type(\cdot)$
Tuple Flatten	$[\![F_A(R)]\!] = \{\{(t.M \circ t.A)^k \mid t^k \in R\}\}$	$\{\{\tau_{R.M} \circ \tau_A\}\}$ where $\tau_{R.A} = \langle A : \tau_A \rangle$ and $M = \text{SCH}(R) - \{A\}$
Relation Flatten	$[\![F_A(R)]\!] = \{\{(t.M \circ u)^{k \cdot l} \mid t^k \in R \wedge u^l \in t.A\}\}$	$\{\{\tau_{R.M} \circ \tau_A\}\}$ where $\tau_{R.A} = \langle A : \{\{\tau_A\}\} \rangle$ and $M = \text{SCH}(R) - \{A\}$
Tuple Nesting	$[\![\mathcal{N}^T_{A \to C}(R)]\!] = \{\{(t.M \circ \langle C : t.A \rangle)^k \mid t^k \in R\}\}$	$\{\{\tau_{R.M} \circ \langle C : \tau_{R.A} \rangle\}\}$ where $M = \text{SCH}(R) - \{A\}$
Relation Nesting	$[\![\mathcal{N}^R_{A \to C}(R)]\!] = \{\{(t.M \circ ns(R,A,C,t))^1 \mid t \in gr(R,A)\}\}$ $gr(R,A) = \{t.A \mid t^n \in R\}, ns(R,A,C,t) = \langle C : [\![\pi_A(\sigma_{A=t}(R))]\!] \rangle$	$\{\{\tau_{R.M} \circ \langle C : \{\{\tau_{R.A}\}\} \rangle\}\}$ where $M = \text{SCH}(R) - \{A\}$
Aggregation	$[\![\gamma_{f(A) \to B}(R)]\!] = \{\{(t \circ \langle B : f(t.A) \rangle)^k \mid t^k \in R\}\}$	$type(R) \circ \{\{\langle B : type(f(A)) \rangle\}\}$

Table 2. Evaluation semantics and output types for the operators of our nested relational algebra for bags.

cost of an exponential blow-up of the problem size. Thus, for performance reasons, it is still important to directly support these question types to avoid this blow-up. However, from a theoretical perspective, the expressive power of these why-not question types is not larger than the fully specified tuple case. In the present paper, we will limit the discussion to fully specified tuples and leave the efficient handling of questions with placeholders to future work.

4.2 Reparameterizations and Explanations

As mentioned in Section 2, the definition of *picky* operators from [5] was shown to fall short in several regards. Here, we take a more principled approach by considering a set of operators as an explanation if there exists a *minimal* (in a precise sense to be defined below) *repair* Q_{repair} of the query which changes this set of operators and for which the missing answer is in the result of $Q_{repair}(D)$. Note that we only identify operators as potential causes instead of actually repairing the query to limit the complexity of the problem. Furthermore, we would like to point out that, for larger instances and/or complex queries, we cannot expect the user to specify all missing answers that should be produced by a query. In this case, a repair that only returns the subset of the missing answers provided by the user is unlikely to produce all missing answers. Thus, we argue that it is more important to identify parts of the query that may be incorrect rather than returning a repair that is likely to be too restrictive.

To formalize such explanations, we have to decide what modifications to the input query are admissible repairs, e.g., we disallow repairs that have nothing in common with the original query since such repairs are unlikely to provide any meaningful information about what is wrong with the input query. Here, we focus on a set of repairs that we refer to as **re-parameterizations (RPs)**. That is, modifications of the input query Q where the query structure is preserved but parameters of operators may differ from Q, e.g., in the running example, we may change $\sigma_{time>\text{'10:00'}}$ to $\sigma_{time>\text{'9:00'}}$ but replacing it with a projection is disallowed. For a RP Q', we use $\Delta(Q, Q')$ to denote the set of operators which have different parameters in Q and Q'.

Successful Re-parameterizations (SRs). We use $\text{RE}(\Phi)$ to denote the RPs for a question Φ. A re-parameterization is *successful* if it produces the missing answer. We use $\text{SR}(\Phi)$ to denote the set of successful re-parameterizations for Φ.

Definition 4.3 (Successful Re-parameterizations)**.** Let $\Phi = (Q, D, t)$ be a why-not question. We call $Q' \in \text{RE}(\Phi)$ a successful re-parameterization if $t \in Q'(D)$.

Minimal Successful Re-parameterizations (MSRs). Obviously, explanations should not be derived from SRs that apply unnecessary changes to query Q. Intuitively, a SR Q'' applies unnecessary changes if we can find another SR Q' that only changes a subset of the operators that Q'' changes and for which the effect on the query result is less than Q'''s effect. We formalize this as a partial order $\preceq_\Phi$ over SRs and, then, define MSRs as SRs that are minimal according to $\preceq_\Phi$.

Definition 4.4 (Minimal Successful Re-parameterization)**.** Let $\Phi = (Q, D, t)$ be a why-not question. We call $Q' \in \text{SR}(\Phi)$ *minimal* if $\nexists Q'' \in \text{SR}(\Phi) : Q'' \preceq_\Phi Q'$. We have $Q' \preceq_\Phi Q''$ if:

1. $\Delta(Q, Q') \subseteq \Delta(Q, Q'')$
2. $Q'(D) - Q(D) \subseteq Q''(D) - Q(D)$
3. $Q'(D) \cap Q(D) \supseteq Q''(D) \cap Q(D)$

Explanations. We use $\text{MSR}(\Phi)$ to denote the set of MSRs for Φ. An explanation for Φ according to one of its MSRs Q' is $\Delta(Q, Q')$, i.e., the operators whose parameters need to be changed to produce t.

Definition 4.5. The set of explanations $\mathcal{E}(\Phi)$ for a why-not question Φ is: $\mathcal{E}(\Phi) = \{\Delta(Q, Q') \mid Q' \in \text{MSR}(\Phi)\}$

Example 4.6. Consider a question Φ_1 using tuple t_1 from Example 4.2. Some RPs are (only modified operators are shown) $Q' : \{\sigma_{time>9:00}\}$ and $Q'' : \{\sigma_{time>9:00}, F_{ride.times.end \to time}\}$. Here, $Q' \in \text{SR}(\Phi_1)$ because the structure of Q is preserved and $t_1 \in Q'(D)$. While $Q'' \in \text{SR}(\Phi_1)$, it is not minimal because $Q' \preceq_\Phi Q''$: Q'' unnecessarily changes the second flatten operator, $Q''(D)$ is a superset of $Q'(D)$, and both retain the same tuples from the original input. In fact, Q' is an MSR and, thus, the filter operator in Q is one possible explanation for the missing answer.

5 Implementation & Evaluation

A naive implementation of the model defined in the previous section would likely be computationally prohibitive. We have developed a prototype that employs a relaxed version of the model. It is built on top of a provenance system for nested data in Spark. Our implementation is guaranteed to produce an explanation when $\mathcal{E}(\Phi) \neq \emptyset$. This improves on the state-of-the-art where no solution may be returned.

Implementation. Our prototype captures provenance for Spark's DataFrame API. DataFrames are collections of tuples that roughly conform to our data model from Section 3. It annotates input tuples with a unique identifiers and tracks them through a query expressible in the algebra outlined in Table 2. For that it instruments DataFrame operators to record associations between (nested) input and output

elements as well as tracks schema manipulations while computing the query result to answer why-not questions later.

In our prototype, why-not questions according to Definition 4.1 are expressed as tree-patterns [11]. A tree-pattern allows for (i) addressing schema elements in a flexible manner (e.g. by supporting ancestor-descendant relationships between elements, or placeholders as defined in Definition 4.1), (ii) accessing individual tuples in nested collections, and (iii) supporting multiplicity constraints over elements in nested collections. While all three features improve the convenience of expressing a why-not question, the latter two features further improve the query performance as Spark's built-in means require expensive flattening, nesting, and join operations to evaluate such patterns.

Given a tree-pattern, query result, and collected provenance, our prototype first identifies which parts of the query result schema-matches the tree-pattern. Similar to [4], it employs an "unrenaming" of schema elements manipulated in the program. Essentially, it traces matching output schema elements back to the input schema. This allows the prototype to (partially) match the unrenamed tree-pattern over the source data to identify source tuples that are potentially relevant for explaining a missing answer. Such input tuples are called *compatibles*. Next, the prototype employs forward tracing of the *compatibles* to identify *picky operators*.

Discussion. While our solution may sound similar to [5], it differs in several important aspects. For instance, our prototype utilizes dedicated provenance capture methods for nested data and processes novel types of why-not questions for nested data using tree patterns. Implementation-wise, it adds support for processing tree-patterns to Spark and, in contrast to [5], mostly operates on a provenance representation that consists of identifier mappings. Thus, it can often avoid materialization or dynamic re-computation of intermediate results.

Let us now compare the output of our prototype with our formal semantics defined in Section 4. As mentioned earlier, our protoype finds an answer if an explanation exists. More precisely, when an explanation $\Delta(Q, Q')$ of size one exists, it is certainly returned. If no explanation in $\mathcal{E}(\Phi)$ consists of one operator, the returned sets of picky operators are strict subsets of explanations from the full set of explanations. Towards finding the complete explanations, one avenue for future research is the injection of "fictive" tuples to explore the full program, similarly to ideas presented in [8].

Evaluation. We conduct a preliminary evaluation of our prototype using the *dblp.org* dataset (2GB). We flatten the authors element of all journal articles and filter for database journals (e.g., *VLDB Journal*) and then create a nested bag of the titles of all such articles for each author. As a why-not question, we ask why there is no result that has *Miller* as author and includes an article titled "Creating probabilistic databases from duplicated data". The cause of the missing answer is the filter condition: DBLP uses *"VLDB J."*.

A naive extension of the algorithms from [4, 5] for nested data would consider all tuples that result from flattening as compatible successors, thus failing to return any explanation. However, for our why-not question, co-authors of *Miller* are not compatible. Our prototype returns the correct result by only tracking tuples whose flattened author is *Miller*.

In terms of performance, we run this scenario five times on a machine with one 2.5Ghz quad-core Intel Core i7 CPU and 16GB memory. Executing the query without provenance tracking took 20.7 seconds on average. Computing the why-not provenance took 25.6 seconds which, in this scenario, is only 24% slower than computing the query result.

6 Conclusions

We present a novel formalization of query-based missing answers, apply it to explain missing answers for queries over nested data, and introduce a Spark-based implementation of this technique. In future work, we will investigate extensions of our algorithm and study which guarantees they provide wrt. our formalization of why-not provenance.

Acknowledgements. This research is partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Projektnummer 251654672 – TRR 161 and NSF Award OAC-1640864.

References

[1] Y. Amsterdamer, SB. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. 2011. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *PVLDB* 5, 4 (2011), 346–357.
[2] K. Belhajjame. 2018. On Answering Why-Not Queries Against Scientific Workflow Provenance. In *EDBT*. 465–468.
[3] N. Bidoit, M. Herschel, and A. Tzompanaki. 2015. Efficient Computation of Polynomial Explanations of Why-Not Questions. In *CIKM*. 713–722.
[4] N. Bidoit, M. Herschel, and K. Tzompanaki. 2014. Query-Based Why-Not Provenance with NedExplain. In *EDBT*. 145–156.
[5] A. Chapman and H. V. Jagadish. 2009. Why not?. In *SIGMOD*. 523–534.
[6] J. Nathan Foster, TJ. Green, and V. Tannen. 2008. Annotated XML: queries and provenance. In *PODS*. 271–280.
[7] S. Grumbach and T. Milo. 1993. Towards Tractable Algebras for Bags. In *PODS*. 49–58.
[8] M. Herschel. 2015. A Hybrid Approach to Answering Why-Not Questions on Relational Query Results. *JDIQ* 5, 3 (2015), 10.
[9] M. Herschel, R. Diestelkämper, and H. Ben Lahmar. 2017. A survey on provenance: What for? What form? What from? *VLDB J.* 26, 6 (2017), 881–906.
[10] M. Interlandi, A. Ekmekji, K. Shah, M. Ali Gulzar, S. Deep Tetali, M. Kim, TD. Millstein, and T. Condie. 2018. Adding data provenance support to Apache Spark. *VLDB J.* 27, 5 (2018), 595–615.
[11] J Lu, TW Ling, Z Bao, and C Wang. 2011. Extended XML Tree Pattern Matching: Theories and Algorithms. *TKDE* 23, 3 (2011).
[12] Chaitanya Mishra and Nick Koudas. 2009. Interactive Query Refinement. In *EDBT*. 862–873.
[13] QT. Tran and CY. Chan. 2010. How to ConQueR why-not questions. In *SIGMOD*. 15–26.