To *Not* Miss the Forest for the Trees - A Holistic Approach for Explaining Missing Answers over Nested Data

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

Seokki Lee
University of Cincinnati, USA
lee5sk@ucmail.uc.edu

Melanie Herschel
University of Stuttgart - IPVS, Germany
National University of Singapore, Singapore
melanie.herschel@ipvs.uni-stuttgart.de

Boris Glavic
Illinois Institute of Technology, USA
bglavic@iit.edu

ABSTRACT

Query-based explanations for missing answers identify which operators of a query are responsible for the failure to return a missing answer of interest. This type of explanations has proven useful, e.g., to debug complex analytical queries. Such queries are frequent in big data systems such as Apache Spark. We present a novel approach to produce query-based explanations. It is the first to support nested data and to consider operators that modify the schema and structure of the data (e.g., nesting, projections) as potential causes of missing answers. To efficiently compute explanations, we propose a heuristic algorithm that applies two novel techniques: (i) reasoning about multiple *schema alternatives* for a query and (ii) re-validating at each step whether an intermediate result can contribute to the missing answer. Using an implementation on Spark, we demonstrate that our approach is the first to scale to large datasets while often finding explanations that existing techniques fail to identify.

CCS CONCEPTS

• **Information systems → Data provenance**; **MapReduce-based systems**; **MapReduce languages**; *Semi-structured data.*

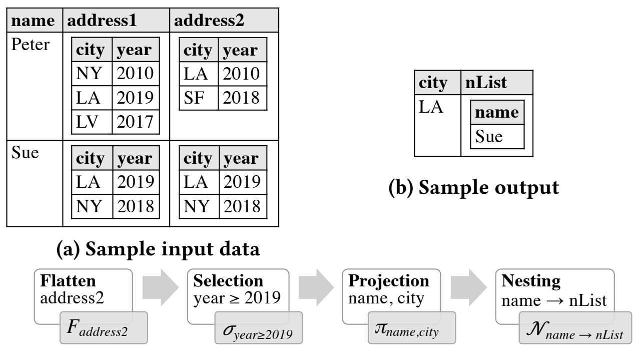
KEYWORDS

query-based explanations; why-not provenance; nested data

ACM Reference Format:
Ralf Diestelkämper, Seokki Lee, Melanie Herschel, and Boris Glavic. 2021. To *Not* Miss the Forest for the Trees - A Holistic Approach for Explaining Missing Answers over Nested Data. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, , 13 pages. https://doi.org/10.1145/3448016.3457249

1 INTRODUCTION

Debugging analytical queries in data-intensive scalable computing (DISC) systems such as Apache Spark is a tedious process. Query-based explanations for missing answers can aid users in this process by narrowing down the debugging task to parts of the query that are responsible for the failure to compute an expected answer. In this work, we present an approach for producing query-based explanations and implement this approach on Spark. Our approach is defined for a nested relational algebra for bags [19]. This allows us to cover a large variety of practical queries expressible in big data systems, like in [1].



name	address1		address2	
	city	year	city	year
Peter	NY	2010	LA	2010
	LA	2019	SF	2018
	LV	2017		
Sue	city	year	city	year
	LA	2019	LA	2019
	NY	2018	NY	2018

(a) Sample input data

city	nList
LA	name
	Sue

(b) Sample output

(c) Operator pipeline for sample program

Figure 1: Given `person` input data (a), we obtain a list of cities with associated persons (b) when running a Spark program that corresponds to the operator pipeline shown in (c).

In general, missing answers approaches have three inputs: a *why-not question* specifying which missing results are of interest, a query, and the input data. Three categories of explanations have been considered [21]: (i) *instance-based* explanations attribute missing answers to missing input data; (ii) *query-based* explanations pinpoint which parts of the query, typically at the granularity of individual operators, cause the derivation of the expected results to fail; and (iii) *refinement-based* explanations produce a rewritten query that returns the missing answer. Our approach returns query-based explanations that consist of *a set of operators*. Each explanation indicates a set of operators that should be fixed for the missing answers to be returned.

Example 1. *Each person tuple in Figure 1a contains two nested address relations (cities with associated years). These may correspond to work and home addresses. Figure 1c shows a query that returns cities that are the workplace of at least one person since 2019. For each such city, the query returns the list of persons that work in this city. The query is composed of four operators (explained further below). The query's result over the person table consists of a single nested tuple (Figure 1b). An analyst may wonder why NY is not in the* *result and pose this concern as a why-not question. Multiple query-based explanations exist. For instance, the selection* `year ≥ 2019` *prevents the tuple (NY, {(Sue)}) that matches the why-not question to appear in the result. Thus, one explanation for the missing answer is that the selection operator needs to be fixed. Another possibility is that the analyst assumed* `address2` *stores work addresses, while in fact,* `address1` *does. However, given the data in* `address1`*, this is not satisfactory to explain the missing answer, as no tuple featuring NY has a sufficiently recent year. Thus, an explanation involving a "misconfigured" flattening operation also requires adjusting the selection, which results in an explanation that includes both operators.*

The idea of providing operators as query-based explanations for a missing answer is at the core of lineage-based approaches [6, 9, 11, 20]. They identify *compatible* tuples in the input data that contain the values necessary to produce the missing answers and trace them through the query to determine *picky operators*. These operators filter *successors* of compatible tuples. The rationale is that it may be possible to change the parameters of a picky operator such that it no longer filters the successors of compatible tuples.

Example 2. *Applying the lineage-based explanation approach to our example for the why-not question asking for NY, we identify tuple (NY, 2018) nested in the* `address2` *attribute of* `Sue` *as the only compatible tuple. When tracing this tuple through the query's operators, we observe that it is in the lineage of the flatten operator's intermediate result. In other words, its successor passes this first operator and is in the input of the subsequent selection. The selection's result does not include any successor of this compatible. Thus, we would identify the selection as a picky operator and return it as an explanation.*

Example 2 already makes non-trivial adaptations to state-of-the-art solutions for relational data. It extends the set of supported operators with flatten and nesting and assumes tracing support for nested tuples. Straightforward extensions of existing solutions would trace top-level tuples only and, thus, return no result at all. More importantly, a purely lineage-based formulation of the problem fails to find all query-based explanations from Example 1.

In this paper, we propose a novel formalization of query-based why-not explanations for both flat and nested data models. We further present a practical algorithm to compute such explanations, which is implemented and evaluated in Apache Spark.

Why-not explanations for flat and nested data based on reparametrizations. Alternative approaches to lineage-based why-not explanations have been investigated recently [5, 10, 14]. However, their practical use is limited since they only support conjunctive queries over relational data or lack an efficient or effective algorithm or implementation. Inspired by [14], our formalization is based on *reparameterizations* of query operators. These are changes to the parameters of one or more operators that "repair" the query such that the missing answer is returned. We define an explanation to be the set of operators modified by a *minimal successful reparameterizations (MSRs)*, which is a reparameterization that is minimal wrt. to a partial order based on the number of operators that are modified (we do not want to modify operators unless needed) and the side effects of the reparameterization ("repairs" should avoid changes to the original query result). Our formalization has two advantages over past work: (i) it guarantees that neither false negatives (operators not returned that have to be changed) nor false positives are returned (operators part of explanations that do not have to be changed); and (ii) explanations may include operators such as projections and nesting (not supported by past work). Such richer explanations require reasoning about the effect that changes to the *schema and (nesting) structure* of intermediate results have on the final query result. However, this precision and expressiveness come at a price: computing MSRs is NP-hard and even restricted cases that are in `PTIME` require further optimizations to be practical.

A scalable heuristic algorithm leveraging schema alternatives and revalidation. In light of this result, we explore a heuristic algorithm that approximates explanations. Given the corners we cut to be efficient, e.g., disregarding reparameterizations of equi-joins to theta-joins that rely on cross products and are of little practical interest in DISC systems, our algorithm may miss certain operators and corresponding MSRs in its returned explanations. Even though our algorithm is heuristic in nature and, like past approaches, uses lineage and forward tracing of compatibles, it often finds explanations they cannot produce. This is due to two novel technical contributions: (i) Our algorithm reasons about multiple *schema alternatives*. It traces changes of the schema and (nesting) structure of intermediate results caused by possible reparameterizations of operators, e.g., flattening `address1` instead of `address2` in our example. (ii) Like previous approaches, it uses *compatibles* to find missing answers. In contrast to them, it *revalidates* compatibility of successors of compatible tuples to avoid false positives (tuples are incorrectly identified as compatible). All past lineage-based approaches are subject to this issue that is exacerbated by considering nested data. For instance, in our example, the complete second input tuple is initially flagged as compatible. After flattening, only one of its two successors is compatible.

Implementation and evaluation. We implement our algorithm in Apache Spark. We highlight design choices that, as experiments validate, make our approach the first to scale to large datasets (we evaluate on datasets several orders of magnitude larger than previous work) and to offer the most expressive query-based explanations to date for both relational and nested data models.

We review related work in Section 2 and introduce preliminaries in Section 3. Our why-not explanations are covered in Section 4. We present our heuristic algorithm in Section 5, our implementation and evaluation in Section 6, and conclude in Section 7.

2 RELATED WORK

Why-not explanations. Most closely related to our work are query-based (e.g., [5, 6, 9–11]) and refinement-based (e.g., [38]) approaches for explaining missing answers. All these approaches target flat relational data and, except for [4, 9], which target workflows, support queries limited to subclasses of relational algebra plus aggregation. As we have seen in the introduction, these approaches do not trivially extend to handling nested data with a richer set of operators and would return fewer explanations than one may expect. This work is an extension of [14].

Query-by-example (QBE) and query reverse engineering (QRE). Query-based explanations for missing answers are also closely related to QBE [12, 13, 43] and QRE techniques [3, 24, 37, 39], which generate a query from a set of input-output examples provided by the user. In contrast to QBE, our explanations start from a given database, query, and output. Opposed to some approaches for

QRE, which return a query equivalent to an unknown query Q, our explanations apply on a given input query that is assumed to be erroneous. Furthermore, in contrast to QBE, QRE, and refinement-based approaches, our approach points out which operators need to be modified rather than returning a complete query.

Query refinement (QR) and the empty answer problem (EAP). QR is also related to our approach [30–32]. QR comes in two forms: relax queries to return more results or contract queries to return fewer results. The former addresses the EAP where a query fails to produce any result, and the latter deals with queries that return too many answers. QR is concerned with quantitative constraints, i.e., how many answer are returned by the query result. In contrast, our work addresses qualitative constraints: the query should return answers with a certain structure and/or content.

Provenance in DISC systems. DISC systems natively support nested data formats such as JSON, XML, Parquet, or Protocol Buffers. Provenance capture for DISC systems has been studied in, e.g., [1, 15, 22, 23, 28, 42]. Why-not explanations are practically relevant in these systems. However, we are not aware of any scalable solution that computes why-not explanations.

Provenance for nested data. Since why-not explanations typically build on the provenance of existing results, our work also relates to work on provenance models for nested data. Like [1, 15, 18], we use a nested data model and query language (a nested relational algebra for bags inspired by [19] in our case).

3 PRELIMINARIES AND NOTATION

3.1 Nested Relational Types and Instances

As in existing work on nested relations [19, 27], we define nested relations as bags (denoted as $\{\{\cdot\}\}$) of tuples. The attributes of a nested relation are either of a primitive type (e.g., booleans or integers), tuples type, or are themselves nested relations.

Definition 1 (Nested Relation Schema). *Let $\mathbb{L}$ be an infinite set of names. A* nested type τ *is an element conforming to the grammar shown below, where each $A_i \in \mathbb{L}$. A type $\mathcal{R}$ is called a* nested relation schema. *A nested database schema $\mathcal{D}$ is a set of $\mathcal{R}$ types.*

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

Definition 2 (Nested Relation Instance). *Let $\mathbb{P}$ denote the domain of primitive type P. We assume the existence of a special value $\perp$ (null) which is a valid value for any nested type. We use* $\mathbf{type}(I)$ *to denote the type of an instance I. The instances I of type τ are defined recursively based on the following rules for primitive types, homogeneous bags, and tuples:* $\frac{I \in \mathbb{P}}{\mathbf{type}(I)=P}$, $\frac{\mathbf{type}(I_1)=\tau,\ldots,\mathbf{type}(I_n)=\tau}{\mathbf{type}(\{\{I_1,\ldots,I_n\}\})=\{\{\tau\}\}}$, $\frac{\mathbf{type}(I_1)=\tau_1,\ldots,\mathbf{type}(I_n)=\tau_n}{\mathbf{type}(\langle A_1:I_1,\ldots,A_n:I_n\rangle)=\langle A_1:\tau_1,\ldots,A_n:\tau_n\rangle}$.

Example 3. *All tuples of the nested relation shown in Figure 1a are of type $\langle name : \textsc{str},\ address1 : \tau_r,\ address2 : \tau_r \rangle$, where τ_r is a nested relation of type $\{\{\langle city : \textsc{str},\ year : \textsc{date} \rangle\}\}$.*

3.2 Nested Relational Algebra

Let R and S denote nested relations, which are manipulated through a nested relational algebra for bags ($\mathcal{NRAB}$). We define $\mathcal{NRAB}$ based on the algebra from [19, 27], which we denote as $\mathcal{NRAB}^0$. $\mathcal{NRAB}^0$ includes operators with bag semantics for selection $\sigma_\theta(R)$, restructuring $map_f(R)$, cartesian product $R \times S$, additive union $R \cup S$, difference $R - S$, duplicate elimination $\epsilon(R)$, and bag-destroy $\delta(R)$. We further define $\mathcal{SPC}$ as the subset of $\mathcal{NRAB}^0$ sufficient to express select-project-join queries, and $\mathcal{SPC}^+$ the algebra that additionally includes additive union to express select-project-join-union queries. These less expressive fragments of $\mathcal{NRAB}^0$ represent the operators commonly supported by lineage-based missing-answers approaches. We use them later for a comparative discussion.

Similarly to [1, 8], we introduce additional operators to ensure a close correspondence between big data programs and the algebra. This is crucial to provide explanations that aid users in debugging their programs. Similarly to [8], we can derive the additional operators from $\mathcal{NRAB}^0$ operators. They include attribute renaming $\rho_{B_1 \leftarrow A_1,\ldots,B_n \leftarrow A_n}(R)$ that renames each attribute A_i of R into B_i, the projection $\pi_{A_1,\ldots,A_n}(R)$ and join variants (i.e., $R \bowtie_\theta S$, $R ⟖_\theta S$, $R ⟕_\theta S$, and $R ⟗_\theta S$), as well as aggregation and variants of nesting and flattening. Together with the operators of $\mathcal{NRAB}^0$, they form our algebra $\mathcal{NRAB}$. Before discussing selected operators of our algebra in more detail, we introduce some notational conventions.

Notation. We denote tuples as $t, t', \ldots$, nested relations as $R, S, \ldots$, and nested databases as $D, D', \ldots$. $\mathcal{R}$ and $\mathcal{D}$ denote the type of a nested relation R and database D, respectively. $t.A$ denotes the projection of tuple t on a set of attributes or single attribute A. $\textsc{sch}(R)$ is the list of attribute names of R. Operator $\circ$ concatenates tuples (types). We also apply $\circ$ to relation types, e.g., $\{\{\langle A : \tau_1 \rangle\}\} \circ \{\{\langle B : \tau_2 \rangle\}\} = \{\{\langle A : \tau_1, B : \tau_2 \rangle\}\}$. We use $t^n \in R$ to denote that t appears in R with multiplicity n and use arithmetic operations on multiplicities, e.g., t^{2+3} means that t appears 5 times. $\textsc{mult}(R, t)$ denotes the multiplicity of t in R. $[\![Q]\!]_D$ denotes the result of evaluating Q over D. We omit D if clear from the context. Finally, $\mathbf{type}(Q)$ denotes the result type of $[\![Q]\!]$. We define some of these operators below. Assume that R is an n-ary input relation of type $\mathcal{R} = \{\{\langle A_1 : \tau_{A_1}, \ldots, A_n : \tau_{A_n} \rangle\}\}$. See [16] for the remaining definitions.

Flatten. The flatten operator unnests the values of an attribute $A \in \textsc{sch}(R)$ which must be of a tuple or relation type. If A is of a tuple type $\tau = \langle \ldots \rangle$, then the **tuple flatten** operator returns a tuple $(t \circ t.A)^k$ for each t^k in R: $[\![F^T_A(R)]\!] = \{\{(t \circ t.A)^k | t^k \in R\}\}$. Its result type is the concatenation of $\mathcal{R}$ and τ: $\mathbf{type}(F^T_A(R)) = \mathcal{R} \circ \{\{\tau\}\}$.

If A is of a nested relation type $\tau = \{\{\langle B_1 : \tau'_1, \ldots, B_m : \tau'_m \rangle\}\}$, then **inner relation flatten** returns each tuple u^l in the nested relation concatenated with the tuple t^k it was initially nested in: $[\![F^I_A(R)]\!] = \{\{(t \circ u)^{k \cdot l} | t^k \in R \wedge u^l \in t.A\}\}$ and $\mathbf{type}(F^I_A(R)) = \mathcal{R} \circ \tau$. We require that none of the attribute names B_i already exist in $\mathcal{R}$.

An **outer relation flatten** behaves similarly to inner relation flatten but additionally returns tuples of R padded with null values if their value of the flattened attribute is the empty relation. That is, using $u_\perp = \langle B_1 : \perp, \ldots, B_m : \perp \rangle$, we define $[\![F^O_A(R)]\!] = F^I_A(R) \ \cup \{\{(t \circ u_\perp)^k \mid t^k \in R \wedge t.A = \emptyset\}\}$.

Nesting. Analogously to the flatten operators, we define two nesting operators: tuple nesting and relation nesting.

Given an attribute set $A \subseteq \textsc{sch}(R)$, **tuple nesting** removes attribute(s) A from each tuple $t \in R$ and adds new attribute C of type τ_A (the tuple type in relation type $\mathbf{type}(\pi_A(R))$) storing $t.A$. Using $M = \textsc{sch}(R) - A$ and τ_M to denote the tuple type of $\mathbf{type}(\pi_M(R))$, we define $[\![N^T_{A \to C}(R)]\!] = \{\{(t.M \circ \langle C : t.A \rangle)^k \mid t^k \in R\}\}$. Accordingly, $\mathbf{type}(N^T_{A \to C}(R)) = \{\{\tau_M \circ \langle C : \tau_A \rangle\}\}$.

Relation nesting $\mathcal{N}^R_{A\to C}(R)$ groups R on M. For each group in $gr(R,M) = \{t.M \mid t^n \in R\}$, the operator returns a tuple with the group-by values ($t.M$) and a fresh attribute C of relation type $\tau_A = \textbf{type}(\pi_A(R))$ that stores the projection of all tuples from the group on A as a nested relation $ns(R,M,A,C,t) = \langle C : [\![\pi_A(\sigma_{M=t.M}(R))]\!]\rangle$. Overall, the result of relation nesting is $[\![\mathcal{N}^R_{A\to C}(R)]\!] = \{\{(t.M \circ ns(R,M,A,C,t))^1 | t \in gr(R,M)\}\}$ with associated type $\textbf{type}(\mathcal{N}^R_{A\to C}(R)) = \{\{\tau_M \circ \langle C : \{\{\tau_A\}\}\rangle\}\}$.
Aggregation. Let $\tau_{in} = \{\{\langle C : \tau\rangle\}\}$. Consider an aggregation function f of type $\tau_{in} \to \tau_{out}$. The **aggregation** operator applies f to the set of values of unary tuples in the results of $\pi_A(R)$ and stores the result in a new attribute B that is of type τ_{out}. Attribute A has to be of type τ_{in}. Thus, $[\![\gamma_{f(A)\to B}(R)]\!] = \{\{(t \circ \langle B : f(t.A)\rangle)^k | t^k \in R\}\}$ and its output type is $\textbf{type}(\gamma_{f(A)\to B}(R)) = \mathcal{R} \circ \{\{\langle B : \tau_{out}\rangle\}\}$.

Example 4. *The operator pipeline of Figure 1c corresponds to the following expression in $\mathcal{NRAB}$:*
$$\mathcal{N}^R_{name\to nList}\left(\pi_{name,city}\left(\sigma_{year\geq 2019}\left(F^I_{address2}(\texttt{person})\right)\right)\right)$$

4 WHY-NOT EXPLANATIONS

We are now ready to formalize the problem of computing why-not explanations for nested (and flat) data.

4.1 Why-Not Questions

A why-not question describes a (set of) expected (nested) tuple(s) that are missing from a query's result $[\![Q]\!]_D$. We let users specify why-not questions as *nested instances with placeholders* (NIPs) that encode *a set of* missing answers, any of which is acceptable to the user. A NIP is a nested instance that in addition to constant values may also contain the *instance placeholder* ? that can stand in for any value and the *multiplicity placeholder* $*$, which can only be used as an element of a nested relation type and represents 0 or more tuples of the nested relation's tuple type. Note that for finite domains, the expressive power of why-not questions with placeholders is not larger than why-not questions based on fully specified tuples. But efficiently supporting the former avoids the exponential blow-up incurred when naively translating them to the latter representation.

Definition 3 (Instances with Placeholders). *Let τ be a nested type. The rules to construct nested instances with placeholders (NIPs) of type τ are: If $\textbf{type}(I) = \tau$ or $I = ?$, then I is a NIP of type τ. Furthermore, if $\tau = \langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle$, then $\langle I_1, \ldots, I_n\rangle$ is a NIP of type τ if each I_i is a NIP of type τ_i. Finally, if $\tau = \{\{\tau_{tup}\}\}$, then $\{\{I_1, \ldots, I_n\}\}$ is a NIP of type τ if (i) $\forall$ I_i either I_i is a NIP of type τ_{tup} or $I_i = *$ and (ii) $\nexists$ $i \neq j \in \{1,\ldots,n\}$ such that $I_i = I_j = *$.*

Example 5. *A NIP that conforms to the output schema of our running example is $t_{ex} = \langle city : NY, nList : \{\{?, *\}\}\rangle$. It stands for all tuples with city equal to NY and at least one name in nList.*

Next, we define the set of nested instances that match a NIP.

Definition 4 (Matching NIPs). *An instance I of type τ matches a NIP I' of type τ, written as $I \simeq I'$ if one of these conditions holds:*
(1) $I' = ?$
(2) $I = I'$
(3) $\textbf{type}(I) = \langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle$ and $\forall i \in [1,n]$, $I.A_i \simeq I'.A_i$
(4) $\textbf{type}(I) = \{\{\tau_{tup}\}\}$ and there exists an assignment $\mathcal{M} \subseteq I \times I' \to \mathbb{N}$ such that all conditions below hold:
  (a) for all $t \in I$ and $t' \in I'$, if $\mathcal{M}(t,t') > 0$ then either $t = t'$, $t' = ?$, or $t' = *$
  (b) for all $t \in I$, $\sum_{t' \in I'} \mathcal{M}(t,t') = \textsc{mult}(I,t)$
  (c) for all $t' \in I'$, either $\sum_{t \in I} \mathcal{M}(t,t') = \textsc{mult}(I',t')$ or $t' = *$

Condition (4) ensures that multiplicies are taken into account.

Example 6. *Consider NIP t_{ex} from Example 5 as well as NIP $t'_{ex} = \langle city : NY, nList : \{\{?, ?\}\}\rangle$. Only the former matches the tuple $t = \langle city : NY, nList : \{\{\langle name : Sue\rangle^2, \langle name : Peter\rangle\}\}\rangle$. Since t is of a tuple type, condition (3) in Definition 4 must hold. While both $t_{ex}.city \simeq t.city$ and $t'_{ex}.city \simeq t.city$ hold (condition (2)), we only have $t_{ex}.nList \simeq t.nList$ (condition (4)). For t'_{ex}, the definition enforces that $\mathcal{M}(\langle name : Sue\rangle, ?) > 0$ and $\mathcal{M}(\langle name : Peter\rangle, ?) > 0$ (condition (4a)) and $\mathcal{M}(\langle name : Sue\rangle, ?) = 2$ (4b). Then, (4c) cannot hold, since the sum is 3 and $\textsc{mult}(t'_{ex}, ?) = 2$. Alternatively assigning each occurrence of $\langle name : Sue\rangle$ to ? causes (4b) to be violated.*

Using NIPs, we now define why-not questions. To ensure that a why-not question asks for a tuple absent from the result, we require that none of the result tuples matches the why-not question's NIP.

Definition 5 (Why-not questions). *Let Q be a query, D a database, and $\textbf{type}([\![Q]\!]_D) = \{\{\tau\}\}$. A why-not question Φ is a triple $\Phi = \langle Q, D, t\rangle$ where why-not tuple t is a NIP of type τ.*

Example 7. *Given D and Q from Figure 1, and the NIP t_{ex} from Example 5, the example why-not question is $\Phi_{ex} = \langle Q, D, t_{ex}\rangle$.*

4.2 Reparameterizations and Explanations

We define query-based explanations for a given why-not question Φ as sets of operators. An explanation is a combination of operators that conjunctively cause tuples matching the NIP t in Φ to be missing from the query result, i.e., it is possible to "repair" the query to return a tuple matching the NIP t (the missing answer) by changing the parameters of these operators. We refer to such repairs as *successful reparameterizations.* The set of explanations produced for a why-not question should consist of sets of operators changed by successful reparameterizations. However, we do not want to return explanations that require more changes than strictly necessary. That is, we want explanations to be minimal in terms of the set of operators they include and in terms of their "side effects" (changes to the original query result beyond appearance of missing answers) a reparametrization of an explanation's operators would have. Existing lineage-based definitions, which generally support queries in $\mathcal{SPC}^+$, do not fulfill our desiderata: (i) They suffer from possibly incomplete explanations (false negatives) [6, 9, 20], i.e., changing the operator they return as an explanation may not be sufficient for returning the missing answer. This motivated alternative definitions [5, 10, 14], albeit limited to conjunctive queries in $\mathcal{SPC}$. (ii) They only reason about operators that prune data (explanations only contain selections and joins) and miss causes at the schema level (e.g., projecting the wrong attribute). (iii) They disregard side effects (which have been considered for instance-based and refinement based explanations [21]). Our formalization addresses all these drawbacks for queries in the rich algebra $\mathcal{NRAB}$.

Our formalization is based on **reparameterizations (RPs)**. A RP for an input query Q is a query Q' that is derived from Q by altering the parameters of operators while preserving the query

Operator op	$param(Q, op)$	**Admissible parameter changes**
Selection $\sigma_\theta(R)$, with θ including attribute references, comparison operators ($=, >, \geq, <, \leq, \neq$ }), and constant values	$\{\theta\}$	Replacing (i) an attribute reference with another attribute from R of the same type; (ii) a comparison operator with another; and (iii) a constant with another constant of the same type.
Restructuring $map_f(R)$	$\{f\}$	Change f
Projection $\pi_L(R)$	$\{A_i \mid A_i \in L\}$	Any substitution of an attribute A_i with an attribute A_j from R
Renaming $\rho_{B_1 \leftarrow A_1, \ldots, B_n \leftarrow A_n}(R)$	$\{(B_1 \leftarrow A_1, \ldots, B_n \leftarrow A_n)\}$	Changing the output attributes based on a permutation of $(B_1, \ldots, B_n)$
Join variants $R \diamond_\theta S$, where $\diamond \in \{\bowtie, \leftouterjoin, \rightouterjoin, \fullouterjoin\}$	$\{\theta, \mathbf{type}(op)\}$, where $\mathbf{type}(op) = \diamond$	(i) Changing the type of join; (ii) replacing a reference to an attribute A with a different attribute B in θ; (ii) replacing a comparison operator with one of $\{=, >, \geq, <, \leq, \neq\}$.
Flatten variants $F^\diamond_A(R)$, where $\diamond \in \{T, I, O\}$	$\{A, \mathbf{type}(op)\}$, where $\mathbf{type}(op) = \diamond$ distinguishes tuple flatten, relation inner flatten, and relation outer flatten	(i) Replacing A by an attribute B in R of tuple type for $\diamond = T$ or relation type otherwise, (ii) changing the flattening type from inner flatten to outer flatten or vice versa
Nesting variants $\mathcal{N}^R_{A \to C}(R)$ or $\mathcal{N}^T_{A \to C}(R)$	$\{A, C\}$	(i) Changing the attributes to be nested / grouped-on (A) or (ii) the name of the attribute storing the result of nesting (C)
Aggregation $\gamma_{f(A) \to B}(R)$	$\{A, B, f\}$	(i) Changing the aggregation function f, (ii) the attribute that we are aggregating over (A), or (iii) the name of the attribute storing the aggregation result (B)

Further parameter-free $\mathcal{NRAB}$ operators are: additive union $R \cup S$, difference $R - S$, deduplication $\epsilon(R)$, cartesian product $R \times S$, bag-destroy $\delta(R)$, and table access R

Table 1: Admissible parameter changes of $\mathcal{NRAB}$ operators.

structure (no operators are added or removed). For instance, changing $\sigma_{year \geq 2019}$ to $\sigma_{year \geq 2018}$ in our running example is a RP, but substituting the selection with a projection is not. We made the choice to preserve query structure to avoid explanations that do not provide meaningful information about errors in the input query.

Table 1 summarizes all admissible parameter changes for all $\mathcal{NRAB}$ operators. They are motivated by what we consider errors commonly arising in practice. Nonetheless, our formalism also applies to alternative definitions of valid parameter changes. However, the choice of allowed parameter changes affects the compuational complexity of the problem (see Section 4.3).

Definition 6 (Valid Parameter Changes). *Given an operator $op \in Q$ with parameters $param(Q, op)$ and a set of predefined admissible parameter changes for this operator type (Table 1), a valid parameter change applies one admissible change to $param(Q, op)$.*

Based on the parameter changes, we define reparameterizations.

Definition 7 (Reparameterizations). *Given a query Q, a query Q' is a reparameterization of Q if it can be derived from Q using a sequence of valid parameter changes.*

For the ease of presentation, we assign each operator $op \in Q$ a unique identifier. Since Q and Q' have same structure, we further assume that an operator $op \in Q$ retains its identifier in Q'. Next, we relate RPs to a why-not question. RPs are **Successful reparameterizations (SRs)** if they produce the missing answer.

Definition 8 (Successful Reparameterizations). *Let $\Phi = \langle Q, D, t\rangle$ be a why-not question and $\text{Re}(Q)$ denote the set of RPs for a query Q, we define $SR(\Phi)$, the set of successful RPs for Q and D, as*

$$SR(\Phi) = \{Q' \mid \exists t' \in [\![Q']\!]_D, t' \simeq t \wedge Q' \in \text{Re}(Q)\}$$

Example 8. *Figure 2 shows a tree representation of nested relations (introduced here as these will become relevant later). The tree T_1 in Figure 2a corresponds to the result $[\![Q]\!]_D$ in our example (Figure 1b). The example why-not question asks why city NY with associated names is missing from $[\![Q]\!]_D$. One possible SR (SR_σ) changes the selection predicate (e.g., to year $\geq$ 2018). This SR produces the result T_2 in Figure 2b. Another SR ($SR_{F\sigma}$) modifies the selection and changes the flattened attribute to address1. It yields tree T_3 (Figure 2c). Additional SRs exist, e.g., changing the year to anything lower than 2018. However, they result in additional changes to $[\![Q]\!]_D$.*

The example above illustrates our rationale to not consider all SR(Φ) as explanations:. (i) some SRs may apply unnecessary changes to Q (e.g., why change both the selection and flatten operator when one is enough?) and (ii) some SRs may cause more changes to the query result than others (e.g., the side effects caused by a less restrictive selection). Figure 2 shows that these two goals (minimizing changes to operators and minimizing side effects) may conflict. Green nodes indicate data matching the why-not tuple, orange nodes mark data not machting the why-not tuple. While T_2 has an entirely orange tuple $\langle city : SF, nList : \{\{\langle name : Peter\rangle\}\}\rangle$, T_3 only has an additional name Peter in attribute $nList$ for LA. Thus, SR_σ changes only a subset of $SR_{F\sigma}$'s operators, but SR_σ entails "more significant" changes to the data (T_2) than $SR_{F\sigma}$ (T_3). To strike a balance between changes to the query and to the data, we define a partial order $\preceq_\Phi$ over SRs and define **minimal successful reparameterizations (MSRs)** as SRs that are minimal wrt. to $\preceq_\Phi$.

Definition 9 (MSRs). *Let $\Phi = \langle Q, D, t\rangle$ be a why-not question and Q', Q'' be two SRs. Let $\Delta(Q, Q')$ denote the set of identifiers of operators whose parameters differ between Q and Q', i.e., $\Delta(Q, Q') = \{op \mid param(Q, op) \neq param(Q', op)\}$. Let d be a distance function quantifying the distance between two nested relations. We define a partial order $Q' \preceq_\Phi Q''$ as follows:*

(1) $\Delta(Q, Q') \subseteq \Delta(Q, Q'')$ *(2)* $d([\![Q]\!]_D, [\![Q']\!]_D) \leq d([\![Q]\!]_D, [\![Q'']\!]_D)$

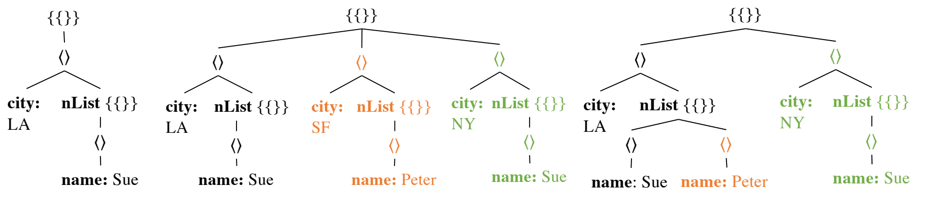
We call $Q' \in SR(\Phi)$ minimal *if $\neg\exists Q'' \in SR(\Phi) : Q'' \preceq_\Phi Q'$.*

We consider operators corresponding to MSRs as **explanations**.

Definition 10 (Explanations). *Let Φ be a why-not question and $MSR(\Phi)$ be the set of MSRs for Φ. We define the set of explanations $\mathcal{E}(\Phi)$ with respect to Φ as $\mathcal{E}(\Phi) = \{\Delta(Q, Q') \mid Q' \in MSR(\Phi)\}$.*

Example 9. *SR_σ and $SR_{F\sigma}$ from Example 8 are both MSRs, because, even though $\Delta(Q, Q'_\sigma) \subseteq \Delta(Q, Q'_{F\sigma})$, we established that $d([\![Q]\!], [\![Q'_\sigma]\!]) > d([\![Q]\!], [\![Q'_{F\sigma}]\!])$, so $Q'_\sigma \not\preceq_\Phi Q'_{F\sigma}$ (and vice versa). We now highlight why we define query-based explanations even though refinement-based explanations are not far fetched given reparameterizations. Assuming we had n address attributes (not just 2), there would be equally many refinement-based explanations involving the flatten operator, some also modifying the selection, others not. A developer would need to understand similarities and differences of all of these before settling on how to fix the query. In contrast, query-based explanations identify sets of operators that need to be fixed.*

The MSR definition leaves the choice of distance function d open. To equally support nested and flat data, a good fit is the tree edit distance for unsorted trees [7, 34]. However, it is NP-hard [41]. Considering an alternative `PTIME` distance metric d will not necessarily result in an efficient algorithm for computing explanations, because, as discussed next, computing explanations is NP-hard in general.



(a) T_1 (b) T_2, result for SR_σ (c) T_3, result for $SR_{F\sigma}$

Figure 2: Tree representations of the result from Figure 1b (T_1) and the results of SRs

```
Algorithm 1: Why-Not(Φ)
1  $\langle \mathcal{M}_{sbt}, \overline{T} \rangle \leftarrow$ schemaBacktracing($\Phi$)
2  $\mathcal{S} \leftarrow$ schemaAlternatives($\mathcal{M}_{sbt}, \overline{T}, \Phi$)
3  $R^A, \leftarrow$ dataTracing($\mathcal{S}, \Phi$)
4  $\mathcal{E}^{\approx} \leftarrow$ computeExplanations($R^A, \mathcal{S}, \Phi$)
5  return $\mathcal{E}^{\approx}$
```

Algebra	Lineage-based	Reparameterization-based
$\mathcal{SPC}$	$\sigma_\theta{}^*, \bowtie_\theta$	$\sigma_\theta{}^*, map_f{}^*, \bowtie_\theta, \pi_L$
$\mathcal{SPC}^+$	$\sigma_\theta{}^*, \bowtie_\theta$	$\sigma_\theta{}^*, map_f{}^*, \bowtie_\theta, \pi_L$
$\mathcal{NRAB}$	$\sigma_\theta{}^*, \setminus^*, \bowtie_\theta, ⟕_\theta, ⋉_\theta, ⟗_\theta, F^I_A(R)$	$\sigma_\theta, map_f, \bowtie_\theta, ⟕_\theta, ⋉_\theta, , ⟗_\theta, \rho_{B_1 \leftarrow A_1, \ldots, B_n \leftarrow A_n}, F^T_A(R), F^I_A(R), F^O_A(R), \mathcal{N}^T_{A \rightarrow C}(R), \mathcal{N}^R_{A \rightarrow C}(R), \gamma_{f(A) \rightarrow B}(R)$

Table 2: Which operators can be part of explanations for which approach? $\mathcal{NRAB}^0$ operators are marked with $*$.

4.3 Discussion

First, we demonstrate that computing explanations for why-not questions is generally NP-hard in terms of data complexity for queries in $\mathcal{NRAB}^0$ (see [16]). We observe that the problem is sensitive to the choice of admissible parameter changes. While it is intractable for the parameter changes shown in Table 1, we identify restrictions of Table 1 for which the problem is in PTIME.

THEOREM 1. *Given a why-not question $\Phi = \langle Q, D, t \rangle$ and a set e of operators from the query Q, testing the membership of e in $\mathcal{E}(\Phi)$ is NP-hard in the size of D for queries that only consist of the operators aggregation, map, projection, renaming, and join. The problem is in* PTIME *if the map operator is restricted to being a projection and if aggregation functions are restricted to the default ones in SQL.*

PROOF SKETCH. We prove the hardness claim for queries with aggregation through a reduction from set cover and sketch a brute force algorithm for the PTIME result. See [16] for the full proof. □

The algorithm we present in Section 5 restricts aggregation and does not consider map. Thus, according to Theorem 1, the problem is in PTIME. However, the search space is still much too large, requiring additional heuristic optimizations to scale.

Next, we discuss differences between reparameterization-based explanations and lineage-based explanations (e.g., [9]). Table 2 shows which operators of $\mathcal{SPC}$, $\mathcal{SPC}^+$, and $\mathcal{NRAB}$ can be identified as causes by these two approaches. Lineage-based solutions generally support $\mathcal{SPC}^+$. They only return operators that remove compatible input data. Thus, for operators overlapping with $\mathcal{NRAB}^0$, only selections become part of explanations. Given that the join operator can be expressed using cross product and selection, lineage-based approaches may return joins as explanations. In our reparameterization-based formalism, the set of operators that can be part of an explanation is already more diverse for the least expressive query class $\mathcal{SPC}$, e.g, we may return projections. The benefits are even more pronounced for $\mathcal{NRAB}$ (last row).

Finally, note that the operators in an explanation depend on a query's algebraic translation and explanations may differ for equivalent translations. For example, $\sigma_\theta(R \times S)$ may only yield the selection while $R \bowtie_\theta S$ may only yield the join. Lineage-based and reparameterization-based solutions share this property.

5 COMPUTING EXPLANATIONS

We now present an algorithm that restricts admissible parameter changes to achieve PTIME data complexity. Furthermore, we introduce novel heuristics that are necessary for efficiency in practice. The algorithm takes a why-not question $\Phi = \langle Q, D, t \rangle$ as input and returns a set of explanations $\mathcal{E}^{\approx}$ that approximates $\mathcal{E}$. We first present the algorithm and, then, discuss in Section 5.5 how $\mathcal{E}^{\approx}$ relates to $\mathcal{E}$ according to Definition 10. A core concept of our approach is *schema alternatives* (SAs). An SA represents a set of RPs that all change attribute references in the query in the same way. For instance, one SA for our running example is to replace *address*1 with *address*2. The RPs corresponding to this SA differ in how (and if) they modify the condition of the query's selection. Our approach determines for each SA a set of NIPs (one per table accessed by the query) such that for any SR corresponding to the SA, only tuples matching these NIPs may contribute to the missing answer. The purpose of these NIPs is to exclude irrelevant data early-on.

Algorithm 1 shows the four main steps of our algorithm. First, given the missing tuple t that is defined over the output schema of Q, the algorithm computes a set of NIP tuples $\overline{T}$ over the schema of Q's input tables in D. It also computes a mapping $\mathcal{M}_{sbt}$ which associates each attribute in t and each attribute referenced in an operator of Q with a set of attributes from the input. We refer to these input attributes as *source attributes.* $\mathcal{M}_{sbt}$ and $\overline{T}$ represent the SA which does not change any attributes. In the second step, Algorithm 1 determines alternatives for each source attribute in $\mathcal{M}_{sbt}$. These alternatives account for attributes that may not have been chosen appropriately when writing Q. The alternative attributes allow the algorithm to enumerate the set of SAs denoted as $\mathcal{S}$. For each SA, we also compute the NIPs $\overline{T}$ for filtering irrelevant input tuples as described above. In the third step (*dataTracing*), we construct a single query that computes the results of the input query under all SAs. Since computing the results for all RPs corresponding to an SA is infeasible, we settle for a more practical alternative: we only compute results for one representative RP for an SA, the one that only changes attribute references in operators according to the SA and nothing else. However, these RPs may not be successful. To be able to reason about other RPs corresponding to a SA, we instrument the query to propagate annotations that encode sufficient information for reasoning about other RPs. For example, instead of filtering tuples that do not match a selection's condition (under some SA), we use an annotation attribute to record which tuples are filtered out by the selection under the representative RP for an SA. The *computeExplanations* function leverages these annotations to compute explanations for Φ ranked according to the partial order of Definition 9 (approximated for performance).

5.1 Step 1: Schema backtracing

Taking the why-not question $\Phi = \langle Q, D, t \rangle$ as input, schema backtracing analyzes schema dependencies and schema transformations of the query Q in a data-independent way. It has two goals: (i) rewrite the missing answer t into a set of NIPs $\overline{T}$ (Definition 3) over the schema of D. We refer to the SA that does not change any attributes as the *base SA*. $\overline{T}$ contains one NIP for each input relation R in D such that all tuples of R that could produce t under some RPs of the base SA match the NIP; (ii) identify attributes from D's schema that serve as alternatives to attributes in Q's input.

To achieve the first goal, we iterate over the query's operators and analyze each operator's parameters to trace data dependencies at the schema level. Eventually, schema backtracing returns $\overline{T} = \{\overline{t}_{R_1}, \ldots, \overline{t}_{R_n}\}$, where R_1 through R_n are Q's input relations. Each $\overline{t}_{R_i}$ is a NIP. The set of tuples from R_i matching $\overline{t}_{R_i}$ includes all tuples that may contribute to tuples matching t under the base SA.

Example 10. *In our running example, one such NIP is* $\overline{t}_{person} = \langle name :?, address1 :?, address2 : \{\{\langle city : NY, year :?\rangle\}\}\rangle$ *computed from* $t = \langle city : \text{“NY”}, nList : \{\{?, *\}\}\rangle$. *To obtain the NIP, the algorithm traces back both dependencies for* $t.city$ *and* $t.nList$. *When it iterates through the operator* $\mathcal{N}^{R}_{name \rightarrow nList}$, *it traces the nested tuples in nList back to the name attribute. The algorithm identifies the name attribute of the person relation as the name attribute's origin. Similarly, it traces city back to the source attribute* $address2.city$, *whose value has to match NY.*

$\overline{T}$ is coupled with a mapping $\mathcal{M}_{sbt}$. This mapping associates each attribute $t.A$ of the why-not tuple t with source attributes to identify the source attributes that produce the values of $t.A$. To also identify source attributes potentially relevant for operator reparameterizations (the second goal outlined above), the backtracing algorithm further adds associations for each attribute reference $op.A$ at operator op to $\mathcal{M}_{sbt}$ while it iterates through the query tree. Notationwise, we distinguish the two types of associations: (i) Associations between a source attribute $\overline{t}.X$ and a missing-answer attribute $t.A$ are denoted as $\frac{A}{X}$. (ii) Associations between a source attribute $\overline{t}.X$ and an operator attribute $op.A$ are written as $\frac{op.A}{X}$. In the following, we represent a pair $(\overline{t}, \mathcal{M}_{sbt})$ as a single nested tuple mirroring the nesting structure of $\overline{t}$ but using associations from $\mathcal{M}_{sbt}$ as attribute names instead. For instance, if $\frac{A}{X}$, $\frac{op.A}{X}$, and $\frac{op.B}{X}$, then we substitute X with $\frac{A, op.A, op.B}{X}$.

Example 11. *Continuing with Example 10,* $\mathcal{M}_{sbt}$ *associates* $t.nList$ *to* $\overline{t}_{person}.name$ *and* $t.city$ *to* $\overline{t}_{persons}.address2.city$. *It also associates* $\sigma.year$ *with* $\overline{t}_{person}.address2.year$. *The associations in* $\mathcal{M}_{sbt}$ *coupled with* $\overline{T} = \{\overline{t}_{person}\}$ *are represented as:*

$$\overline{t}_{person} = \Big\langle \frac{t.nList,\ \pi.name,\ N.nList,\ N.name}{name} :?, address1 :?,$$
$$\frac{F.address2}{address2} : \{\{\langle \frac{t.city, \pi.city}{city} : \text{“NY”}, \frac{\sigma.year}{year} :?\rangle\}\}\Big\rangle$$

5.2 Step 2: Schema alternatives

Next, the algorithm determines schema alternatives (SAs), which potentially produce the missing answer based on reparameterizations implementing these SAs. A SA substitutes zero or more attributes in operator parameters with alternatives. The set of all SAs covers all such substitutions.

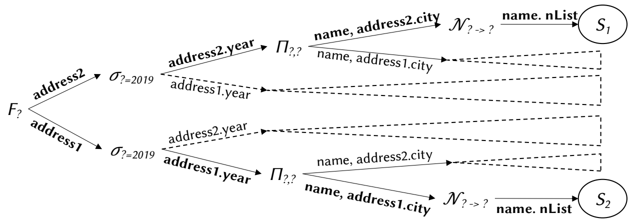


Figure 3: Enumerating and pruning schema alternatives

Finding attribute alternatives. The first step of identifying SAs is to find alternatives for attributes referenced by Q. For each $\overline{t}_{R_i} \in \overline{T}$, we identify, for each $\frac{A}{X} \in \mathcal{M}_{sbt}$ a set of alternative attributes $\mathcal{X}' = \{X'_1, \ldots X'_k\}$ such that $X'_j \in R_i$ and $\textbf{type}(X'_j) = \textbf{type}(X)$. We restrict alternatives to attributes of the same relation, because replacing an attribute with an attribute from another relation would require more changes to the query than reparametrizations allow. We assume that the set of attribute alternatives is provided as input to our algorithm. For instance, these can be determined by hand, through schema matching techniques [2, 17], or using schema-free query processors [25, 26]. These strategies ensure that we only consider meaningful alternatives.

Enumerating and pruning SAs. The attribute alternatives are used to enumerate all possible SAs, which requires considering alternatives for attributes of intermediate results that appear as $\frac{op.A}{X} \in \mathcal{M}_{sbt}$. Formally, a schema alternative $S = \langle \overline{T}, \mathcal{M} \rangle$ is a set of NIPs $\overline{T}$ (as in schema backtracing, one tuple per table accessed by Q) and a mapping $\mathcal{M}$ (like $\mathcal{M}_{sbt}$, $\mathcal{M}$ records which input attributes are referenced by which operator and are used to derive which attribute in Q's output). As mentioned above, $\overline{T}$ over-approximates the set of tuples that contribute to the derivation of the missing answer under the SRs for the SA.

Example 12. *Consider the following attribute alternatives:* $name'$=$\{name\}$, $city'$=$\{address2.city, address1.city\}$, $year'$=$\{address2.year, address1.year\}$, *and* $address2'$=$\{address2, address1\}$. *Figure 3 shows how the algorithm incrementally derives all SAs (ignore the dashed parts for now). Based on the set of alternatives for* $address2'$ *and* $year'$, *it starts evaluating options for the flatten operator's parameters. We can either use the original attribute* $address2$, *or the alternative* $address1$. *For each alternative for flatten, we can then choose* $address2.year$ *or* $address1.year$ *for the selection operator.*

SAs replace attributes of one operator independently from attributes of another operator. Thus, some SAs may lead to an invalid query that references non-existing attributes in some operators or alter Q's output schema (not allowed). The algorithm prunes these alternatives. For instance, after flattening $address2$, the only “accessible” alternative for $year$ is $address2.year$ in the selection. Further assuming the source data included $address1.city1$ instead of $address1.city$, flattening $address1$ changes Q's output schema to $\{\{\langle city1, nList\rangle\}\}$, which is not allowed.

Example 13. *In our example, all dashed subtrees in Figure 3 are pruned. Only two SAs remain, denoted as* S_1 *and* S_2. *The SA* $S_1 = \langle\{\overline{t}_1\}, \mathcal{M}_1\rangle$, *with* $\overline{t}_1$ *being equal to* $\overline{t}_{person}$ *shown in Example 11, and*

$S_2 = \langle\{\bar{t}_2\}, \mathcal{M}_2\rangle$ with $\bar{t}_2$ "swapping" the address attribute, i.e.,

$$\bar{t}_2 = \langle \frac{t.nList, \pi.name, N.nList, N.name}{name} :?, address2 :?, \frac{F.address1}{address1} : \{\{\langle \frac{t.city, \pi.city}{city} : \text{"NY"}, \frac{\sigma.year}{year} :?\rangle\}\}\rangle$$

5.3 Step 3: Data tracing

At this point, the algorithm has identified the source attributes to consider for reparameterizations ("blue numerators" identified during schema backtracing) and has determined the reparameterizations to consider for attributes (through SAs). Next, it identifies and traces data that may yield the missing answer through reparameterizations of query operators. It instruments operators to compactly keep track of possible reparameterizations and their results.

We define individual tracing procedures for each operator. The procedures commonly take the operator op, an annotated relation R^A, and schema alternatives $\mathcal{S}$ as input. Their output consists of an annotated relation $R^{A'}$ and updated schema alternatives $\mathcal{S}'$. The algorithm extends operator semantics to collect result tuples under all SAs and record information about all RPs corresponding to these SAs in annotation attributes added to the operator's result schema.

We distinguish four annotation types each stored in additional attributes added to each tuple t' in the instrumented operator's output $R^{A'}$.

- id: Each top-level tuple is assigned a unique identifier. Our algorithm leverages the ids to trace fine-grained provenance as in [15]. It utilizes the provenance to correctly maintain tuples throughout tracing and during $\mathcal{E}^{\approx}$ computation.
- $validS_i$: For each SA S_i, this boolean annotation describes whether t' is part of the operator output under SA S_i. Our algorithm leverages this annotation to determine which $t' \in R^{A'}$ correspond to which SA.
- $consistentS_i$: For each SA S_i, this boolean annotation identifies if a tuple t' is consistent with the why-not question. t' is consistent if it potentially contributes to the missing answer under some SR for the SA. This annotation stores the result of re-validating compatibles as hinted at in the introduction.
- $retainedS_i$ indicates if t' is an output tuple of the original query except for attribute changes given by S_i (true) or if it requires additional operator reparameterizations, e.g., changing constants in a selection condition, to exist (false).

In the following, we describe the tracing algorithms for the operators used in our running example, omitting projection since it simply propagates consistent and valid annotations of its input.

Table access. The tracing procedure for the table access operator iterates over each tuple t in the input relation R and extends t with annotation attributes. It adds the id attribute and a $consistentS_i$ attribute for each SA S_i. The value v of this attribute is only true if t matches the tuple $\bar{t}_R$ in the set of tuples $\overline{T}_i$ of S_i. To add correctly named annotations in function of S_i, we use the *annotate* function (Algorithm 2), e.g., we call $annotate(t', [(consistent, v)], S_i, op)$. The table access operator does not change the structure of its input, so input SAs are simply propagated to its output.

Example 14. *Applying the procedure for table access to our running example yields the annotated relation shown in Figure 4. Schema alternative S_1 is associated to $\bar{t}_1$ shown in Example 13 and considers address2.city, while S_2 comprises $\bar{t}_2$ using address1.city. The first tuple in Figure 4 has $consistentS1\_1 = 0$ because it has no value in address2.city that matches $\bar{t}_1$'s constraint city = "NY", while $consistentS2\_1 = 1$ because address1.city nests $\langle city : \text{"NY"}, 2010\rangle$.*

Algorithm 2: *annotate*

```
1 Function annotate(t, avMap, S_i, op):
2     foreach (a, v) ∈ avMap do
3         label ← a+"S"+i+"_"+op.getID()
4         t ← t ∘ ⟨label : v⟩
5     return t
```

name	address1		address2		id_1	consistent S1_1	consistent S2_1
	city	year	city	year			
Peter	NY	2010	LA	2010	1	0	1
	LA	2019	SF	2018			
	LV	2017					
Sue	LA	2019	LA	2019	2	1	1
	NY	2018	NY	2018			

Figure 4: Example of annotations after table access

Algorithm 3: $Flatten(op, R^A, \mathcal{S})$

```
1  Function Flatten(op, R^A, S):
2      ∀S_i ∈ S, let O_i be the result of executing op wrt S_i and generalized to an outer flatten
3      ∀S_i ∈ S, let S'_i = ⟨T̄'_i, M'_i⟩ be the schema alternative reflecting the flattening wrt S_i
4      O_merged ← ∅
5      foreach t ∈ O_1 do
6          r ← t is in the result of original flatten wrt S_1
7          c ← t ≃ t̄'_R, where t̄'_R ∈ T̄'_1
8          avMap ← [(valid, 1), (retained, r), (consistent, c)]
9          O_merged ← O_merged ∪ {annotate(t, avMap, S_i, op)}
10     foreach O_i, 1 < i ≤ |S| do
11         t̄ ← t̄'_R ∈ T̄'_i
12         O_merged ← merge(O_merged, O_i, S_i, op, t̄)
13     return ⟨O_merged, ∪_{S_i ∈ S} S'_i⟩
```

Flatten. The tracing procedure for the flatten operator (Algorithm 3) computes the results of the operator under all schema alternatives. It obtains the result O_i of the outer flatten for each SA S_i. It uses an outer flatten for two reasons. First, changing an inner flatten to an outer flatten is a valid parameter change. Second it has to track tuples that the inner flatten filters because the flattened attribute is null or the empty set. Next, the algorithm updates the SAs to reflect the restructuring of the tuples. It then combines all O_i as follows. Lines 5–9 process O_1, i.e., the result of the outer flatten parameterized as given by S_1. For each tuple t in O_1, it evaluates boolean conditions to determine the values c and r for the *consistent* and *retained* flags. The algorithm sets the *valid* annotation to 1. To process the remaining SAs (lines 10–12), it uses the *merge* function. Intuitively, *merge* concatenates tuples with the same id across the outer flatten results of all SAs, ensuring not to replicate columns that remain the same across all SAs. Since the number of tuples with a given id may vary across the different results (due to nested relations of varying cardinality), it pads missing "concatenation partners" with null values (⊥). The algorithm creates annotations for each SA. Thus, it sets annotations corresponding to null-padded (non-existent) alternatives to 0. The annotations of tuples in each O_i are set analog to the ones for S_1. Each tuple produced in the output also receives a fresh unique id.

Example 15. *Given the annotated relation in Figure 4 and the SAs in Example 13, the inner flatten produces the annotated relation shown in Figure 5 and updates S_1 with $\overline{T}'_1 = \{\langle \frac{nList}{name} :?, \frac{city}{cityS1} : \text{"NY"}, yearS1 :?\rangle\}$ and S_2 analogously. It combines both SAs, as both address1 and address2 are flattened. The column marked with ... summarizes all annotation columns of the input. They are treated as "regular" input columns when executing the outer flatten.*

name	city	year	city	year	…	consistent	retained	valid	consistent	retained	valid	id_2
	S2	S2	S1	S1		S1_2	S1_2	S1_2	S2_2	S2_2	S2_2	
Peter	NY	2010	LA	2010	…	0	1	1	1	1	1	3
Peter	LA	2019	SF	2018	…	0	1	1	0	1	1	4
Peter	LV	2017	⊥	⊥	…	0	0	0	0	1	1	5
Sue	LA	2019	LA	2019	…	0	1	1	0	1	1	6
Sue	NY	2018	NY	2018	…	1	1	1	1	1	1	7

Figure 5: Example of annotations after flatten

Focusing on the new annotations, we see in the column $consistentS1\_2$ that only the last tuple is consistent with $\overline{T}'_1$, because it is the only tuple that features "NY" in $cityS1$. Further, the 1 values in $validS1\_2$ indicate that the flatten produces 4 tuples under S_1. The third tuple is not valid under S_1, being an artifact of unnesting $address1$ for SA S_2. The other tuples all have the $retainedS1\_2 = 1$. Thus, no tuple is lost due to the more restrictive inner flatten type.

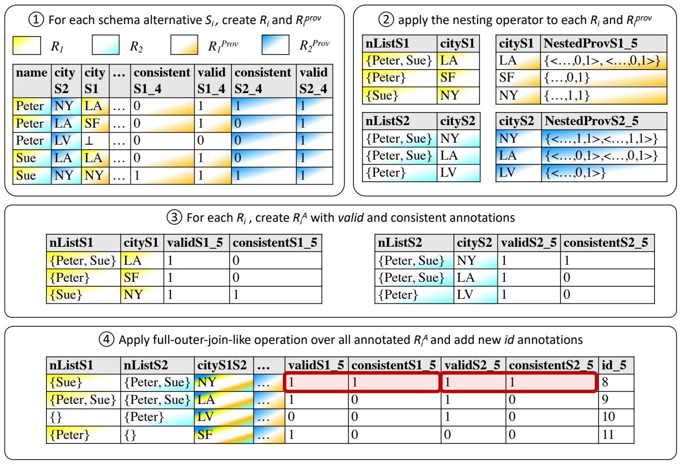
Selection. The tracing procedure for the selection operator returns all input tuples with additional annotation columns. It propagates the $consistent$, $valid$ and id attributes of the previous operator, since it neither manipulates the schema nor the identity of top-level tuples. However, the procedure adds a new $retained$ attribute for each S_i. The value of the $retained$ attributes is 1 if a tuple from the input under S_i satisfies the selection condition θ, and 0 otherwise.

Example 16. *Ignoring the red highlighting for now, Figure 6 shows the tracing output after the selection checking if $year \geq 2019$. For instance, the last tuple has $year = 2018$ under S_1, so $retainedS1\_3 = 0$.*

Relation nesting. Due to space constraints, we explain the algorithm for relation nesting only based on our running example. Since the nesting changes the structure of the input relation, our algorithm first updates the set of SAs. For each S_i, it derives an alternative S'_i from S_i that reflects nesting. This, for instance, yields S'_1 with $\overline{T}'_1 = \{\langle \frac{nList}{nListS1} : \{\{\langle name : ?\rangle, *\}\}\rangle, \frac{city}{cityS1} : \text{"NY"}\}$. Then, the algorithm computes the result of relation nesting considering the schema alternatives and annotates the result tuples as shown in Figure 7. First, for each S_i, it computes R_i by "isolating" all columns involved in schema alternative S_i and retaining valid tuples only. Similarly, S_i yields R_i^{prov} by projecting on all annotation columns related to S_i and selecting valid tuples. Figure 7 ① shows the result of tracing the preceding projection operator. It highlights data of R_1 in yellow and R_2 in cyan, while data of R_1^{prov} and R_2^{prov} are highlighted in orange and dark blue, respectively. In step ②, the algorithm nests R_i and R_i^{prov}. Processing S_1 results in the top row of tables for step ②, while S_2 yields the two bottom relations. Annotations are added to tuples of R_i in step ③, resulting in R_i^A. For all tuples, the valid annotation is set to 1, whereas the consistent annotation is set to 1 only if $t \in R_i$ matches $\overline{t}'_R \in \overline{T}'_i$. For instance, in Figure 7③, the third tuple of R_1 (left) is flagged as consistent, because it matches the constraints defined by $\overline{T}'_1$. Finally, in step ④, all relations R_i^A and R_i^{prov} of all schema alternatives are combined using a function similar to a full outer join. Instead of padding values with nulls when no join partner exists, the algorithm pads the nested relations with $\emptyset$ and the annotations with 0. That allows

name	city	year	city	year	…	consistent	retained	valid	consistent	retained	valid
	S2	S2	S1	S1		S1_3	S1_3	S1_3	S2_3	S2_3	S2_3
Peter	NY	2010	LA	2010	…	0	0	1	1	0	1
Peter	LA	2019	SF	2018	…	0	0	1	0	1	1
Peter	LV	2017	⊥	⊥	…	0	0	0	0	0	1
Sue	LA	2019	LA	2019	…	0	1	1	0	1	1
Sue	NY	2018	NY	2018	…	1	0	1	1	0	1

Figure 6: Example of annotations after selection



① For each schema alternative S_i, create R_i and R_i^{prov}

R_1 R_2 R_1^{Prov} R_2^{Prov}

name	city	city	…	consistent	valid	consistent	valid
	S2	S1		S1_4	S1_4	S2_4	S2_4
Peter	NY	LA	…	0	1	1	1
Peter	LA	SF	…	0	1	0	1
Peter	LV	⊥	…	0	0	0	1
Sue	LA	LA	…	0	1	0	1
Sue	NY	NY	…	1	1	1	1

② apply the nesting operator to each R_i and R_i^{prov}

nListS1	cityS1
{Peter, Sue}	LA
{Peter}	SF
{Sue}	NY

cityS1	NestedProvS1_5
LA	{<…,0,1>, <…,0,1>}
SF	{…,0,1}
NY	{…,1,1}

nListS2	cityS2
{Peter, Sue}	NY
{Peter, Sue}	LA
{Peter}	LV

cityS2	NestedProvS2_5
NY	{<…,1,1>,<…,1,1>}
LA	{<…,0,1>,<…,0,1>}
LV	{<…,0,1>}

③ For each R_i , create R_i^A with *valid* and consistent annotations

nListS1	cityS1	validS1_5	consistentS1_5
{Peter, Sue}	LA	1	0
{Peter}	SF	1	0
{Sue}	NY	1	1

nListS2	cityS2	validS2_5	consistentS2_5
{Peter, Sue}	NY	1	1
{Peter, Sue}	LA	1	0
{Peter}	LV	1	0

④ Apply full-outer-join-like operation over all annotated R_i^A and add new *id* annotations

nListS1	nListS2	cityS1S2	…	validS1_5	consistentS1_5	validS2_5	consistentS2_5	id_5
{Sue}	{Peter, Sue}	NY	…	1	1	1	1	8
{Peter, Sue}	{Peter, Sue}	LA	…	1	0	1	0	9
{}	{Peter}	LV	…	0	0	1	0	10
{Peter}	{}	SF	…	1	0	0	0	11

Figure 7: Example of annotations after relation nesting

the algorithm to compose operators extended with our tracing procedure. It also collapses joined columns (the non-nested attributes) from the different schema alternatives (e.g., $cityS1$ and $cityS2$), by coalescing their values. The final result of step ④ is shown at the bottom of Figure 7 (ignore red highlighted boxes for now).

5.4 Step 4: Computing Explanations

The result of the data tracing step is a nested relation that extends the original query result with (i) data that could belong to the result under some reparameterization and (ii) annotations needed to identify the operators that require a reparameterization to obtain the missing data. Algorithm 4 approximates the set of explanations formally defined in Section 4.2. It first initializes partial explanations E_i based on attribute substitutions imposed by schema alternatives $S_i \in \mathcal{S}$. For instance, in Figure 3, S_1 does not involve any change in the attributes referenced by query operators (and thus, $E_1 = \emptyset$), whereas S_2 involves changing the attribute referenced by the flatten operator (and thus $E_2 = \{F\}$). Then Algorithm 4 initializes a *queue* with pairs of the last operator in Q and the partial explanations E_i for each S_i. Next, it retrieves each operator op_j of the query (top-down) and their associated partial explanations E_i from *queue* to check if op_j needs to be added to an E_i. More precisely, op_j extends E_i when the annotations relative to op_j and the schema alternative S_i contain at least one valid tuple that is consistent with the why-not question, is not retained, and in the lineage of a consistent tuple of the final result. The algorithm further adds op_j's predecessor op_{j-1} with unchanged E_i to the queue when it is possible that explanations without op_j but with some of its predecessors can be found (i.e., all annotations are set to 1 for op_j). When no further operators can be added, it adds E_i to $\mathcal{E}^{\approx}$.

Example 17. *$\mathcal{E}^{\approx}_{example}$ contains two explanations: $E_1 = \{\sigma\}$ and $E_2 = \{F, \sigma\}$ computed from the annotations in red boxes in Figures 6 and 7. These explanations are based on the MSRs SR_1 and SR_2 described in Example 9.*

Currently, we only compute loose upper and lower bounds (UB and LB) for side effects. Obtaining the exact number of side effects would require comparing the original query result to the result of any possible actual reparameterization for each operator. For example, $year \geq 2018$ and $year \neq 2019$ are both possible actual

Algorithm 4: *computeExplanations*$(R^A, \mathcal{S}, \Phi)$

```
Let $E_i$ be the explanation prefix determined for each $S_i \in \mathcal{S}$
Let $op_{last}$ be the final operator in $\Phi.Q$
$queue \leftarrow$ add all pairs $(op_{last}, E_i)_{S_i}$ in the context of $S_i$
$\mathcal{E}^{\approx} \leftarrow \emptyset$
while $queue \neq \emptyset$ do
    $(op_j, E_i)_{S_i} \leftarrow queue.removeFirst()$
    $R^A_{ij} \leftarrow$ annotations relative to $op_j$ and $S_i$
    $extendWithOp \leftarrow false$
    if $R^A_{ij}$ contains a valid tuple $t$ where $retainedSi\_j = 0$ and $consistentSi\_j = 1$
      and $t$ is in the lineage of a consistent output tuple then
        $extendWithOp \leftarrow true$
    if $j > 1$ then
        if $extendwithOp$ then
            $queue.append(op_{j-1}, E_i \cup \{op_j\})$
        if $R^A_{ij}$ contains a valid tuple with all its annotations being set to 1 then
            $queue.append(op_{j-1}, E_i)$
    else
        if $extendwithOp$ then
            $\mathcal{E}^{\approx} \leftarrow \mathcal{E}^{\approx} \cup \{E_i \cup \{op_j\}\}$
        if $R^A_{ij}$ contains a valid tuple with all its annotations being set to 1 then
            $\mathcal{E}^{\approx} \leftarrow \mathcal{E}^{\approx} \cup \{E_i\}$, if $E_i \neq \emptyset$
Prune $\mathcal{E}^{\approx}$ based on upper and lower bounds of side effects for each explanation in $\mathcal{E}^{\approx}$
  and sort them according to the partial order defined in Definition 9.
return $\mathcal{E}^{\approx}$
```

reparameterizations of the selection operator in E_1 and E_2, but may yield a different number of side effects.

We compute $LB = LB(\Delta+)+LB(\Delta^-)$ and $UB = UB(\Delta^+)+UB(\Delta^-)$ based on estimates on the maximum (for UB) and minimum (for LB) number of top-level tuples *any* operator reparameterization in an explanation adds (Δ^+) or removes (Δ^-) from the original query result $[\![Q]\!]_D$. For explanations within the original schema alternative, which we will consistently denote as S_1, $UB(\Delta^+)$ equals the number of valid top-level tuples in the result that have at least one retained flag set to 0 for one of the explanation's operators. For instance, in Figure 7, tuples 9 and 11 satisfy this condition for explanation E_1. For explanations linked to a SA S_i, $i \neq 1$ that does not represent the original query Q, the upper bound is the number of valid top-level tuples with values under S_i different from tuples under S_1 having all their retained and valid flags set to 1, e.g., tuple 9 and tuple 10. $UB(\Delta^-)$ equals $|[\![Q]\!]_D|$ minus the number of valid top-level tuples under the considered SA that match an original tuple (with only true valid and retained flags) under S_1. In our example, all result tuples not matching the why-not question have at least one nested value with a false retained flag, so we get $UB(\Delta^-) = 1$ for both explanations. For explanations involving a selection or join, the lower bound is always set to 0, because we do not know if a reparametrization different from the "full relaxation" of the operator that our tracing algorithms model may avoid the side effects. In all other cases, we estimate $LB(\Delta^+) = max(\text{number of valid and retained tuples} - |[\![Q]\!]_D|, 0)$ and $LB(\Delta^-) = max(|[\![Q]\!]_D| - \text{number of valid and retained tuples}, 0)$. We leave algorithms that compute tighter bounds to future work. Finally, the explanations are ordered following the partial order defined in Definition 9, ranking E_1 higher than E_2.

5.5 Discussion

We observe that our algorithm guarantees that any returned explanation is a correct explanation. However, given our loose bounds on side effects, we cannot guarantee that they all yield MSRs. Furthermore, we may miss some operators / explanations due to the algorithm's heuristic nature. Essentially, the proposed algorithm cuts the following corners for efficiency, causing certain cases not to be accurately covered: (i) It considers only equi-joins and does not model a reparameterization to theta-joins. This avoids cross products that enumerate all possible outputs of join reparameterizations. If such a reparameterization was an explanation, our algorithm misses it. (ii) The tracing procedures for selection, join, and flatten faithfully cover reparameterizations yielding more tuples, compared to the original query operator. So we miss explanations where a more restrictive selection condition, join type, or flatten type would yield a missing answer. (iii) Finally, for aggregations, we generally do not trace the result for different subsets of their input data, which is particularly problematic when selections precede it (for changing equi-join types and flatten types, this is manageable). Also, we do not consider changing the aggregation function.

6 IMPLEMENTATION AND EVALUATION

We implement the algorithm of Section 5 as summarized in Section 6.1. We describe the test setup in Section 6.2. Section 6.3 covers our quantitative evaluation on scalability, while Section 6.4 discusses the quality of returned explanations.

6.1 Implementation

While the concepts apply to DISC systems in general, we implement them in Spark's DataFrame API. DataFrames are tuple collections matching our data model from Section 3. The transformations supported by Spark's DataFrame API can be expressed in our algebra (Section 3.2). To express and process why-not questions (Definition 5), we leverage the tree-patterns implementation from [29].

Our prototype integrates into Spark's query planning and execution phases. The schema backtracing (Section 5.1) and schema alternatives computation (Section 5.2) integrate into the query planning phase. Data tracing (Section 5.3) and computing explanations (Section 5.4) span across both phases. Similar to [33], our prototype rewrites the query plan to directly obtain the explanations from provenance annotations added for data tracing.

A straightforward implementation of data tracing does not result in efficient query plans. We incorporate multiple optimizations to avoid blow-up in query size and avoid cross products. These careful design choices make our algorithm scale to dataset sizes several orders of magnitude larger than those any other state-of-the-art solution can handle. At the same time, we produce many explanations that lineage-based approaches do miss.

6.2 Test Setup

We test on a Spark 2.4 cluster with 50 executors of 16GB RAM each. We define 16 scenarios on three nested datasets: T1 to T4 and T$_{ASD}$ (the latter adapted from [36]) on Twitter data, D1 to D5 on DBLP data, and 6 scenarios on a nested version of TPCH that nests lineitems into orders [35] with queries corresponding mostly (as explained later) to the benchmark queries Q1, Q3, Q4, Q6, Q10, and Q13 without the unsupported sorting and top-k selection. The Twitter dataset consists of tweets with roughly 1000 mostly nested attributes [40]. DBLP contains records of different

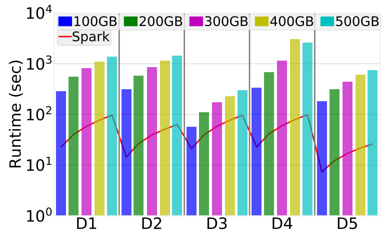


Figure 8: Runtime for DBLP

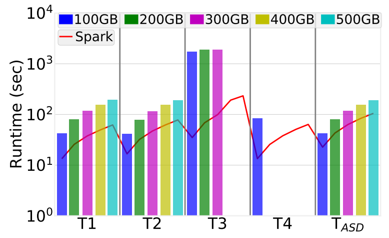


Figure 9: Runtime for Twitter

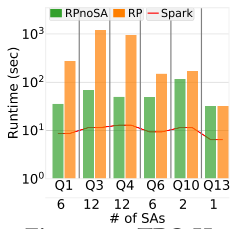


Figure 10: TPC-H

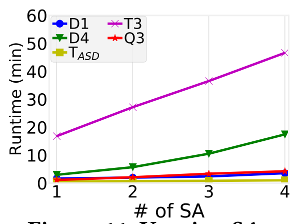


Figure 11: Varying SAs

types, such as article, author, etc. Table 3 summarizes our scenarios. For each scenario, it provides a short description and highlights its query operators (ignore the rest for now). By default, each Twitter and DBLP scenario has 2 schema alternatives (SAs), i.e., the basic SA plus one SA using an attribute alternative. For the TPCH scenarios, we identify three sets of attribute alternatives: (i) $\{l\_discount, l\_tax\}$, (ii) $\{l\_shipdate, l\_commitdate, l\_receiptdate\}$, and (iii) $\{o\_orderpriority, o\_shippriority\}$. This can result in up to 12 SAs, depending on the attributes used by a query. Additional details including the queries in $\mathcal{NRAB}$ and why-not questions are provided in [16].

When not mentioned otherwise, we apply a scale factor of 10 for TPCH and consider 100GB of DBLP or Twitter data. To evaluate runtime and scalability, we vary the DBLP and Twitter dataset size between 100GB and 500GB. To assess explanation quality, we deliberately modified operators in the T_{ASD} and TPCH queries. The unmodified queries serve as a gold standard, such that the explanations precisely containing the modified operators are the correct ones. We study the explanations returned by our reparameterization-based algorithm with (**RP**) and without (**RPnoSA**) multiple schema alternatives. We further compare these to the explanations of a lineage-based approach **WN++**. To this end, we extended Why-Not [9] to scale to big data and to support nested data.

6.3 Performance Evaluation

Varying dataset size. The bars in Figures 8 and 9 report RP's runtime for DBLP and Twitter scenarios for varying dataset sizes given a 2 hours time-out. The line reports the original query runtime.

First, we note linear scalability with the input size. Second, our implementation exceeds the runtime of the original query by a factor between 2.4 and 78.2, depending on the scenario. This overhead is in line with the overhead of state-of-the art solutions on relational data. This overhead is particularly low for queries with a low number of operators, such as D3, T2, and T_{ASD}. The overhead increases when the queries become more complex (D4, D5, T3, T4). For such queries, our annotations grow in size, causing additional runtime overhead and even exceeding our time-out limit for larger input sizes of T3. Furthermore, joins are expensive. Spark rewrites the joins in D4 and T3 from Hash-Joins to much slower Sort-Merge-Joins, since it does not support outer Hash-Joins. However, we require the outer joins to accurately trace tuples without a join partner. Moreover, high runtime overhead occurs when the output is based on a small subset of the input tuples. For example, in D5, two inner flatten operators on nested relations that are empty for most tuples yield much fewer output tuples than input tuples. In contrast, our tracing algorithm retains at least one output tuple for each input tuple. Finally, for T4, we only show results for 100GB input data, because we did hit a Spark limitation for larger sizes. It is related to a reported bug in Spark's grouping set implementation, which we use in the aggregation tracing procedure, and Spark's current item limit in nested collections (2^{31}).

For the TPCH scenarios (Figure 10) the overhead was between a factor of 3.9 and 10.1 for RPnoSA, and up to 105.2 for RP. It is higher for two reasons. First, all TPCH queries use aggregations. Thus, their result size is insignificant compared to the number of traced tuples (analogous to D5). Second, the higher numbers of SAs cause higher overhead (up to 12 compared to 2 for DBLP and Twitter).

Varying the number of SAs. To study the runtime impact of SAs, we consider between 1 and 4 SAs. We report results for a simple scenario with only a few operators and insignificant changes in intermediate result sizes (T_{ASD}), two scenarios of intermediate difficulty with relation flatten and join operators (D1, T3), and two difficult scenarios featuring flatten, join, nesting, and aggregation (D4, Q3). Figure 11 shows the results. For all but the most difficult scenarios, the runtime increases by a constant factor of 0.15 (T_{ASD}), 0.5 (D1), or 0.8 (T3) per added SA. Since the factor is below 1, adding an SA to the rewritten query is faster than executing seperate queries for each SA. In D4, adding SAs causes some slow down. While the factor is 0.96 for adding the first alternative, it is 1.47 for adding the last alternative. Similarly, for Q3, going up to 12 SAs has an overhead of 4.76x compared to 4 SAs and 17.92 compared to one SA. The reason is twofold. First, with each added SA, each tuple's size increases. Second, the grouping set implementation in Spark used for our aggregations duplicate each input tuple for each alternative. Thus, both the tuple width and the tuple number increase with each SA, explaining observed runtimes.

6.4 Explanation Quality

We summarize the explanations returned by WN++, RPnoSA, and RP for all scenarios in Table 3. Flag ○ denotes that all algorithms did find an explanation involving this operators. The flag ◑ indicates that WN++ misses an explanation involving an operator of this type, which both RPnoSA and RP find. Operators appearing only in RP's explanations are marked ●. ◓ denotes that WN++ did produce incomplete explanations, i.e., explanations that involve the marked operator, but require modifying another operator that WN++ misses. 🔴 denotes incorrect explanations only found by WN++. Note that some cells have multiple flags, because a query may contain multiple operators (with different id) of the same type. As shown in the three rightmost columns in Table 3, WN++ finds 12, RPnoSA detects 21, and RP yields 48 explanations. The numbers in brackets behind the number of explanations indicate the position of the correct explanations for the scenarios with a gold standard.

Next, we describe scenarios Q3, Q10, and T_{ASD}. We use op^{id} to distinguish multiple operators of the same type (see [16] for the queries in $\mathcal{NRAB}$). Scenario Q3 computes unshipped orders. We have introduced a typo in the constant commitdate in σ^{27} and

Scen.	Query	Operators						# explanations		
		σ	π	$\bowtie$	F	$\mathcal{N}$	γ	WN++	RPnoSA	RP
D1	All authors and titles of papers that are published at SIGMOD	○	●					1	1	2
D2	Number of articles for authors who do not have "Dey" in their name				●			0	0	1
D3	Lists all author-paper-pairs per booktitle and year					●		0	0	1
D4	Papers per author that have published through ACM after 2010	○ ◑			●			1	2	4
D5	List of (hompage) urls for each author		●		○			1	1	2
T1	List of tweets providing media urls about a basketball player	◑			◓ ◑ ●			1	1	2
T2	All users who tweeted about BTS in the US	○ ◑			●			1	2	4
T3	Hashtags and medias for users that are mentioned in other tweets				○ ●			1	1	2
T4	Nested list of countries for each hashtag of tweets containing “UEFA”	◓ ◑			●			1	1	3
T_{ASD}	ASD example [36]: flatten, filter, project quoted tweets (2 modifications)	●			●			0 (-)	0 (-)	2 (2)
Q1	TPCH query 1 with one modified aggregation	○					●	1 (-)	1 (-)	3 (2)
Q3	TPCH query 3 with two modified selections	◓ ◑ ●					●	1 (-)	1 (1)	2 (1)
Q4	TPCH query 4 with a modified selection and aggregation	●					●	0 (-)	0 (-)	4 (3)
Q6	TPCH query 6 with one modified selection	○ ◑ ●						1 (-)	7 (2)	11 (2)
Q10	TPCH query 10 with two modified selections and a modified projection	◑ ●	●	🔴				1(-)	2(-)	4 (4)
Q13	TPCH query 13 with one modified join			○				1 (1)	1(1)	1 (1)

○ : Found by all algorithms, ◑ : found only by RPnoSA and RP, ● : found only by RP, ◓ : WN++ is incomplete 🔴 WN++ is incorrect

Table 3: Summary of explanations returned for the lineage-based approach WN++, our reparameterization-based approach without SAs (RPnoSA) and our fully fledged approach RP. Shaded fields indicate that a scenario's query uses one or more operators of this type, and the shaded circles indicate operators found by the different approaches (see legend).

replaced the marketsegment in σ^{26} as errors and miss a certain order in the output. WN++ finds only σ^{27} as an explanation, because it removes the order entirely from its output. Unlike our solution, WN++ misses that σ^{26} on the marketsegment would also remove the missing order. Thus, RPnoSA, and RP return $\{\sigma^{26}, \sigma^{27}\}$ as their first and correct explanation. RP also returns explanation $\{\sigma^{26}, \sigma^{27}, \gamma^{25}\}$. It is based on schema alternatives reflecting the tax as alternative to the discount. The last explanation yields the missing order since the order appears in the result regardless of the SA. This scenario shows that our solution already outperforms WN++ without SAs.

Scenario Q10 reports returned items and the associated revenue loss. We introduce three errors in the query. We replace the constants in the selections σ^{35} on the returnflag and σ^{36} on the orderdate. Additionally, we substitute the discount with the tax in the projection π^{37} that computes the discount on the correct, non-zero revenue. We expect a missing customer in the result who generates noticable revenue. WN++ returns the join $\bowtie^{38}$ on customer and order as an explanation because it removes the expected customer from the result. While the explanation makes the customer appear in the result, it cannot yield a non-zero revenue, which we did ask for. Thus, this explanation is incorrect. RPnoSA and RP first point to σ^{35} since it removes all potential join partners for the expected customer. Next, RPnoSA and RP return both selections $\{\sigma^{35}, \sigma^{36}\}$ because σ^{36} also removes tuples that join with the expected customer. RP further returns $\{\sigma^{35}, \pi^{37}\}$ and $\{\sigma^{35}, \sigma^{36}, \pi^{37}\}$, which add π^{37} to the discussed explanations. The last explanation is based on SAs and precisely points at all our modifications. It is ranked last, since it modifies the most operators. However, note that one would have obtained the correct solution iteratively when observing the provided selections before the projection. Our solution does not return $\bowtie^{38}$ since it cannot yield a non-zero revenue.

We finally describe the adaptive schema database (ASD) scenario T_{ASD} from [36]. An ASD extracts and refines relational schemata from semi- or unstructured data. T_{ASD} extracts one relation each for the nested retweeted tweets, and the nested quoted tweets. To extract the retweeted tweets the ASD (i) flattens them with F^{21}, (ii) filters a non-null retweet count in σ^{22}, and (iii) projects only the attributes from the retweet. To reflect the ambiguity between retweets and quotes, we add two errors to T_{ASD}. We flatten the quoted tweets and filter on the quote count. The missing answer is a certain retweet. As finding these errors requires SAs, only RP finds explanations, i.e., $\{F^{21}\}$ and $\{F^{21}, \sigma^{22}\}$. T_{ASD} shows that RP adds two key features to ASDs. It helps resolving schema ambiguities through SAs and finding missing data in the output relations.

In general, even RPnoSA finds explanations that WN++ misses (T1, T4, Q3, Q6, Q10) because RPnoSA traces through the entire query. While these results are for nested data, WN++ exhibits the same problem for flat relational data. as described in [16]. Furthermore, RP may find explanations based on SAs that both RPnoSA and WN++ miss (as in all scenarios except Q13). In fact, the SAs may be the only means to obtain an explanation at all (D2, D3, T_{ASD}, Q4). When multiple operators need reparameterizations, our solution provides the correct explanation, but possibly not ranked at the top, like in Q10 and T_{ASD}. However, the operators of higher ranked explanations typically intersect with the operators in the correct explanation. Thus, starting investigations with the higher-ranked explanations seems a viable option to incrementally correct a query.

7 CONCLUSIONS

We present a novel approach for query-based explanations for missing answers that is the first to (i) support nested data, (ii) consider changes to the query that affect the schema of intermediate results, and (iii) scale to big data (100s of GBs). Even for queries over flat data, which prior work is limited to, it produces explanations that existing systems miss. One avenue for future research is to define and efficiently compute tighter bounds for side effects.

Acknowlegements. Partially funded by Deutsche Forschungsgemeinschaft (DFG) under Germany's Excellence Strategy - EXC 2075 - 390740016 and by NSF grants OAC-1640864 and IIS-1956123.

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. *Proceedings of the VLDB Endowment (PVLDB)* 5, 4 (2011), 346–357.

[2] David Aumueller, Hong-Hai Do, Sabine Massmann, and Erhard Rahm. 2005. Schema and Ontology Matching with COMA++. In *ACM Conference on the Management of Data (SIGMOD)*.

[3] Pablo Barceló. 2019. A Theoretical View on Reverse Engineering Problems for Database Query Languages. In *International Workshop on Description Logics*, Mantas Simkus and Grant E. Weddell (Eds.), Vol. 2373.

[4] K. Belhajjame. 2018. On Answering Why-Not Queries Against Scientific Workflow Provenance. In *Conference on Extending Database Technology (EDBT)*. 465–468.

[5] N. Bidoit, M. Herschel, and A. Tzompanaki. 2015. Efficient Computation of Polynomial Explanations of Why-Not Questions. In *Conference on Information and Knowledge Management (CIKM)*. 713–722.

[6] N. Bidoit, M. Herschel, and K. Tzompanaki. 2014. Query-Based Why-Not Provenance with NedExplain. In *Conference on Extending Database Technology (EDBT)*. 145–156.

[7] Philip Bille. 2005. A survey on tree edit distance and related problems. *Theoretical computer science* 337, 1-3 (2005), 217–239.

[8] Jesús Camacho-Rodríguez, Dario Colazzo, Melanie Herschel, Ioana Manolescu, and Soudip Roy Chowdhury. 2016. Reuse-based Optimization for Pig Latin. In *Conference on Information and Knowledge Management (CIKM)*.

[9] A. Chapman and H. V. Jagadish. 2009. Why not?. In *ACM Conference on the Management of Data (SIGMOD)*. 523–534.

[10] Daniel Deutch, Nave Frost, Amir Gilad, and Tomer Haimovich. 2018. NLProveNAns: Natural Language Provenance for Non-Answers. *Proceedings of the VLDB Endowment (PVLDB)* 11, 12 (2018), 1986–1989.

[11] Daniel Deutch, Nave Frost, Amir Gilad, and Tomer Haimovich. 2020. Explaining Missing Query Results in Natural Language. In *Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 427–430.

[12] Daniel Deutch and Amir Gilad. 2019. Reverse-Engineering Conjunctive Queries from Provenance Examples. In *Conference on Extending Database Technology (EDBT)*. 277–288.

[13] Gonzalo Diaz, Marcelo Arenas, and Michael Benedikt. 2016. Sparqlbye: Querying RDF data by example. *Proceedings of the VLDB Endowment (PVLDB)* 9, 13 (2016), 1533–1536.

[14] Ralf Diestelkämper, Boris Glavic, Melanie Herschel, and Seokki Lee. 2019. Query-based Why-not Explanations for Nested Data. In *International Workshop on Theory and Practice of Provenance (TaPP)*.

[15] Ralf Diestelkämper and Melanie Herschel. 2020. Tracing nested data with structural provenance for big data analytics. In *Conference on Extending Database Technology (EDBT)*. 253–264.

[16] Ralf Diestelkämper, Seokki Lee, Melanie Herschel, and Boris Glavic. 2021. To not miss the forest for the trees - A holistic approach for explaining missing answer over nested data (supplementary material). (2021). arXiv:2103.07561 [cs.DB]

[17] Hong Do and Erhard Rahm. 2002. COMA - A System for Flexible Combination of Schema Matching Approaches.. In *Conference on Very Large Data Bases (VLDB)*. 906 – 908.

[18] J. Nathan Foster, TJ. Green, and V. Tannen. 2008. Annotated XML: queries and provenance. In *Symposium on Principles of Database Systems (PODS)*. 271–280.

[19] S. Grumbach and T. Milo. 1996. Towards Tractable Algebras for Bags. *Journal of Computer and System Sciences (JCSS)* 52, 3 (1996), 570 – 588.

[20] M. Herschel. 2015. A Hybrid Approach to Answering Why-Not Questions on Relational Query Results. *ACM Journal on Data and Information Quality (JDIQ)* 5, 3 (2015), 10.

[21] M. Herschel, R. Diestelkämper, and H. Ben Lahmar. 2017. A survey on provenance: What for? What form? What from? *The VLDB Journal* 26, 6 (2017), 881–906.

[22] R. Ikeda, H. Park, and J. Widom. 2011. Provenance for Generalized Map and Reduce Workflows. In *Conference on Innovative Data Systems Research (CIDR)*. 273–283.

[23] 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. *The VLDB Journal* 27, 5 (2018), 595–615.

[24] Dmitri V Kalashnikov, Laks VS Lakshmanan, and Divesh Srivastava. 2018. FastQRE: Fast Query Reverse Engineering. In *ACM Conference on the Management of Data (SIGMOD)*. 337–350.

[25] Yunyao Li, Cong Yu, and H. V. Jagadish. 2004. Schema-Free XQuery. In *Conference on Very Large Data Bases (VLDB)*. 72–83.

[26] Yunyao Li, Cong Yu, and H. V. Jagadish. 2008. Enabling Schema-Free XQuery with meaningful query focus. *The VLDB Journal* 17, 3 (2008), 355–377.

[27] Leonid Libkin and Limsoon Wong. 1997. Query Languages for Bags and Aggregate Functions. *Journal of Computer and System Sciences (JCSS)* 55, 2 (Oct. 1997), 241–272.

[28] D. Logothetis, S. De, and K. Yocum. 2013. Scalable lineage capture for debugging DISC analytics. In *Symposium on Cloud Computing (SOCC)*. 17.

[29] J Lu, TW Ling, Z Bao, and C Wang. 2011. Extended XML Tree Pattern Matching: Theories and Algorithms. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 23, 3 (2011).

[30] Chaitanya Mishra and Nick Koudas. 2009. Interactive Query Refinement. In *Conference on Extending Database Technology (EDBT)*. 862–873.

[31] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. 2008. Generating Targeted Queries for Database Testing. In *ACM Conference on the Management of Data (SIGMOD)*. New York, NY, USA, 499–510.

[32] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. 2016. A Holistic and Principled Approach for the Empty-answer Problem. *The VLDB Journal* 25, 4 (2016), 597–622.

[33] Tobias Müller, Benjamin Dietrich, and Torsten Grust. 2018. You Say 'What', I Hear 'Where' and 'Why'? (Mis-)Interpreting SQL to Derive Fine-Grained Provenance. *Proceedings of the VLDB Endowment (PVLDB)* 11, 11 (2018), 1536–1549.

[34] Mateusz Pawlik and Nikolaus Augsten. 2011. RTED: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment (PVLDB)* 5, 4 (2011), 334–345.

[35] P. Pirzadeh, M. Carey, and T. Westmann. 2017. A performance study of big data analytics platforms. In *Conference on Big Data*. 2911–2920.

[36] William Spoth, Bahareh Sadat Arab, Eric S. Chan, Dieter Gawlick, Adel Ghoneimy, Boris Glavic, Beda Christoph Hammerschmidt, Oliver Kennedy, Seokki Lee, Zhen Hua Liu, Xing Niu, and Ying Yang. 2017. Adaptive Schema Databases. In *Conference on Innovative Data Systems Research (CIDR)*.

[37] Wei Chit Tan, Meihui Zhang, Hazem Elmeleegy, and Divesh Srivastava. 2017. Reverse Engineering Aggregation Queries. *Proceedings of the VLDB Endowment (PVLDB)* 10, 11 (2017), 1394–1405.

[38] QT. Tran and CY. Chan. 2010. How to ConQueR why-not questions. In *ACM Conference on the Management of Data (SIGMOD)*. 15–26.

[39] Quoc Trung Tran, Chee Yong Chan, and Srinivasan Parthasarathy. 2014. Query Reverse Engineering. *The VLDB Journal* 23, 5 (2014), 721–746.

[40] Zhiyi Wang and Shimin Chen. 2017. Exploiting Common Patterns for Tree-Structured Data. In *ACM Conference on the Management of Data (SIGMOD)*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 883–896.

[41] Kaizhong Zhang, Rick Statman, and Dennis Shasha. 1992. On the editing distance between unordered labeled trees. *Information processing letters* 42, 3 (1992), 133–139.

[42] N. Zheng, A. Alawini, and Z. G. Ives. 2019. Fine-Grained Provenance for Matching ETL. In *IEEE International Conference on Data Engineering (ICDE)*. 184–195.

[43] M.M. Zloof. 1977. Query-by-Example: A Data Base Language. *IBM Systems Journal* 16, 4 (1977), 324–343.