Wondering why data are missing from query results? Ask Conseil Why-Not

Melanie Herschel

Wondering Why Data are Missing from Query Results? Ask Conseil Why-Not

Melanie Herschel
Université Paris Sud 11 / INRIA Saclay Ile-de-France
91405 Orsay Cedex
melanie.herschel@lri.fr

ABSTRACT

In analyzing and debugging data transformations, or more specifically relational queries, a subproblem is to understand why some data are not part of the query result. This problem has recently been addressed from different perspectives for various fragments of relational queries. The different perspectives yield different, yet complementary explanations of such *missing-answers*.

This paper first aims at unifying the different approaches by defining a new type of explanation, called *hybrid* explanation, that encompasses the variety of previously defined types of explanations. This solution goes beyond simply forming the union of explanations produced by different algorithms and is shown to be able to explain a larger set of missing-answers. Second, we present *Conseil*, an algorithm to generate hybrid explanations. *Conseil* is also the first algorithm to handle non-monotonic queries. Experiments on efficiency and explanation quality show that *Conseil* is comparable to and even outperforms previous algorithms.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms

Keywords

why-not data provenance, query-analysis

1. INTRODUCTION

In designing data transformations, e.g., for data integration tasks, developers often face the problem that they cannot properly inspect or debug the individual steps of their transformation specification, which is commonly specified declaratively. Instead, when observing result data that do not match their expectation, developers manually search for the reason for the unexpected behavior.

One important sub-problem in this context is the explanation of *missing-answers*, i.e., data missing from the query result (although the developer expected it). Recently, approaches to explain missing-answers of relational and SQL queries have been proposed. These approaches generate either *instance-based explanations* [12, 13], *query-based explanations* [4], or *modification-based explanations* [18], which we illustrated by the following example.

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.

http://dx.doi.org/10.1145/2505515.2505725.

```
SELECT P.ProdID, Name, MAX(Rating)
FROM Ratings R, Products P
WHERE R.ProdID = P.ProdID
AND P.Loc = 'US'
GROUP BY P.ProdID, P.Name
HAVING MAX(R.Rating) <= 2
```

Products

ProdID	Name	Loc
P1	Car	US
P2	Truck	US
P3	Bus	CH

Ratings

ProdID	Rating
P1	5
P2	2
P2	1

Figure 1: Sample query and input data

EXAMPLE 1. *Fig. 1 shows an SQL query and sample input data. We assume ProdID is a primary key in Products.*

Assume the tuple $\langle$P1, Car, $v_1\rangle$ is not in the query result, although the developer or an analyst expected it to be. Here, v_1 is a variable standing for "could be any value". An instance-based explanation for this missing-answer may indicate that the maximum rating of the product with ProdID = P1 exceeds 2, i.e., Ratings "wrongly" includes one or more tuples of the form $\langle$P1, $v_2\rangle$, where v_2 is a variable value that is required to be above 2 (in denoting such conditional tuples in the future, we will add the condition as last attribute, e.g., $\langle$P1, v_2, $v_2 > 2\rangle$). A query-based explanation may identify that, although the product exists in Products together with corresponding ratings in Ratings, the selection predicate MAX(R.Rating) $\leq$ 2 is responsible for filtering the missing-answer. Finally, a modification-based explanation modifies the query such that the tuple appears in the result, e.g., it may raise the selectivity of the selection by changing it to MAX(R.Rating) <= 5.

Unfortunately, it is not guaranteed that, given a missing-answer, an algorithm finds an explanation. As the next example shows, it is even possible that no explanation of any type is returned.

EXAMPLE 2. *Continuing our example, no explanation can be computed for the missing-answer $\langle$P3, Bus, 0$\rangle$. Indeed, an instance-based explanation would have to insert tuple $\langle$P3, Bus, US$\rangle$ into Products (in addition to inserting the missing rating), which is however not possible due to the constraint on ProdID [12, 13][1]. Due to the lack of a rating of 0 in Ratings, neither a query-based explanation nor a modification-based explanation will be computed by state-of-the-art algorithms [4, 18] that assume the existence of data necessary to produce the missing-answer in the source tables.*

Ideally, an explanation pointing out both the problem of missing source data and the problem of problematic query operators would help a developer in analyzing the query in the above example. Therefore, we introduce a novel type of explanation that combines existing types of explanations and produces an explanation even in cases where no other individual approach produces a result. We refer to this new type of explanation as *hybrid explanation*.

[1][13] also considers updating attribute values but we can easily exclude this solution by enforcing a trust constraint [13].

EXAMPLE 3. *In the scenario of Ex. 2, a possible hybrid explanation inserts a tuple ⟨P3, 0⟩ into Ratings so as to fulfill the join with the existing tuple in Products. In addition, it points out that this combination of source data does not make it to the result because of the selection predicate on location.*

To generate hybrid explanations, we present the *Conseil* algorithm. More specifically, we provide a formal **definition of hybrid explanations** and **extend** definitions of other types of explanations to also support **non-monotonic** queries. We also present the ***Conseil* algorithm** that computes hybrid explanations for a restricted class of **non-monotonic queries** and **experimentally compare it** to state-of-the art algorithms.

In the rest of this paper, we first analyze related work in Sec. 2. Next, we formalize our framework to compute hybrid explanations in Sec. 3. Sec. 4 focuses on the *Conseil* algorithm, which is evaluated in Sec. 5 before we conclude in Sec. 6. A long version of this paper is available at http://nautilus.saclay.inria.fr/publications.

2. RELATED WORK

The problem of simplifying the analysis of the behavior of data transformations to more easily understand and verify transformation behavior and semantics has been addressed by a variety of techniques [5, 9, 7, 11, 15, 17]. The work presented in this paper falls in the category of data provenance research [5], focusing on a specific sub-problem referred to as why-not provenance [4, 16]. We briefly review existing approaches for why-not provenance.

The Missing-Answers (MA) algorithm [13] computes instance-based explanations given a single missing tuple and a single select-project-join (SPJ) query. Essentially, it rewrites the SPJ query such that the result of the rewritten query corresponds to all possible instance-based explanations for the specified missing-answer. Instance-based explanations either insert or update the source data, and their number can be reduced by trusting tables (attributes), which prevents inserts (updates) on these.

Artemis [12] extends the MA algorithm in the sense that it applies to a set of non-nested SQL queries that involve selection, projection, join, union, and aggregation (SPJUA queries). Furthermore, more than one missing-answer can be specified. Artemis also considers explanation side-effects for pruning explanations. A side-effect is any tuple that, upon changing the source data according to an instance-based explanation, appears in the result of any considered query in addition to the specified missing-answer.

Why-Not [4] computes query-based explanations. First, given a missing-answer, it identifies tuples in the source database that contain the constant values or that satisfy the conditions of the missing-answer and that are not part of the lineage [6] of any tuple in the query result. The values in those tuples are traced over the query operators to identify which operators have them as input but not as output. In [4] the algorithm is shown to work for one query involving selection, projection, join, and union (SPJU query).

ConQueR [18], outputs modification-based explanations. Given a set of missing-answers, an SPJUA query, and a source database, it first determines if the necessary source data to produce the missing-answers are available. This is similar to Why-Not. The SQL query is then changed such that all missing-answers become part of the output, while side-effects are minimized.

The algorithm in [10] computes modification-based explanations while considering side-effects to answer why-not questions on top-k queries [10]. Its focus lies on changing k or preference weights to make the missing-answer appear in the query result.

Compared to previous work, *Conseil* is the first to consider non-monotonous queries and hybrid explanations. However, it does not consider side-effects nor top-k queries.

3. FRAMEWORK AND DEFINITIONS

In defining our unified framework for why-not provenance, we first define the most general input scenario, referred to as *debugging scenario* (as in [12]). Our definition, relying on conditional tuples [14], covers the general case that considers multiple missing-answers and multiple queries and provably captures all previous definitions while offering enough freedom for further algorithms.

DEFINITION 1 (CONDITIONAL TUPLE (C-TUPLE)). *A conditional tuple $t = \langle a_1, \ldots, a_n, cond\rangle$ is a tuple with attributes a_1 to a_n having constant or variable values, and $cond$ being a boolean expression. The semantics are that tuple t represents all possible tuples that contain the same constants and that satisfy $cond$.*

To indicate the relation R a c-tuple t = $\langle a_1, \ldots, a_n\rangle$ belongs to, we use $R\langle a_1, \ldots, a_n, cond\rangle$, and $relation(t) = R$. Also, we refer to an attribute a within a c-tuple t using the notation $t.a$.

DEFINITION 2 (DEBUGGING SCENARIO). *A debugging scenario is a 5-tuple $(E, \mathcal{Q}, \mathcal{Q}(D), D, \mathcal{C})$, where $\mathcal{Q}$ is a set of queries, $\mathcal{Q}(D)$ is the result of these queries over some source instance D, E is a set of missing-answers to be explained, specified as a set of c-tuples missing from $\mathcal{Q}(D)$, and $\mathcal{C}$ being a set of constraints defined over the remaining four components of the debugging scenario.*

Let us now turn to the definition of explanations, i.e., the output of an algorithm explaining missing-answers. Our definitions generalize previous definitions of query-based, instance-based, and modification-based explanations to cover queries involving operators from relational algebra plus aggregation.

An instance-based explanation consists of labeled c-tuples. The definition of these relies on compatible c-tuples.

DEFINITION 3 (COMPATIBLE C-TUPLES). *A c-tuple t_1 is compatible with a c-tuple t_2 if (i) $\Pi_{a_1,\ldots,a_n}(t_1)$ is equal, subsumes, or complements $\Pi_{a_1,\ldots a_n}(t_2)$ and (ii) t_1 or the complement of t_1 and t_2 satisfies $t1.cond \wedge t_2.cond$.*

We reuse existing definitions for subsumption and complementation [2, 8], except that unlike these, we consider value NULL as part of the constant domain (and NULL equals NULL!), and unknown semantics are attributed to the variables of a c-tuple.

EXAMPLE 4. *Consider c-tuples $t_1 = \langle$P1, Car, v_1, $v_1 \neq$ 'UK'$\rangle$, $t_2 = \langle$P1, v_2, US, v_2 LIKE 'C%'$\rangle$, and $t_3 = \langle$P1, Car, US, true$\rangle$. Here, t_3 subsumes t_1 because t_3 matches all constants of t_1 and has less unknown values and t_3 satisfies the condition of $t_1.cond \wedge t_3.cond$. Thus, t_3 is compatible with t_1 (but not vice versa). Focusing on t_1 and t_2, we see that these complement each other. The complement of t_1 and t_2 (without conditions) is $\langle$P1, Car, US$\rangle$ for which it is easy to verify that both $t_1.cond$ and $t_2.cond$ hold. Hence, t_1 is compatible with t_2 (and vice versa).*

DEFINITION 4 (LABELED C-TUPLE). *A labeled c-tuple $t = L\langle a_1, \ldots, a_n,$ cond$\rangle$ w.r.t. some data set D is a c-tuple associated with a label $L \in \{+, -, \circ\}$ that indicates whether a c-tuple compatible with t is known to exist in D ($L = \circ$), needs to exist in D ($L = +$), or must not exist in D ($L = -$).*

When associated with label $\circ$, the c-tuple describes an existing tuple and hence its condition is always true. For brevity, we omit the condition $true$ for tuples in D in the remainder of this paper.

DEFINITION 5 (INSTANCE-BASED EXPLANATION). *An instance-based explanation ϕ_{IB} for a debugging scenario describes modifications to the database D that would yield the missing-answers of E in $\mathcal{Q}(D)$ while satisfying constraints $\mathcal{C}$. The syntax of ϕ_{IB} is:*

$\phi_{IB} := \{[\mathcal{T}_1, \ldots \mathcal{T}_n]\}_A$ $\quad \mathcal{T} := C|\phi_{IB}$
$C := L\langle a_1, \ldots, a_n, cond\rangle$ $\quad A := group|agg|group \wedge agg|\emptyset$
$group := group|acopv$ $\quad agg := agg|aggF(a)aCond$

where $L\langle a_1, \ldots, a_n, cond\rangle$ refers to Def. 4, a is an attribute, v a value (constant or variable), $cop \in \{=, <, >, \leq, \geq\}$, $aggF$ is an aggregation function over attribute a, and $aCond$ a condition on the aggregated value of a. The semantics of ϕ_{IB} describe the sequence of operations $[\mathcal{T}_1, \ldots, \mathcal{T}_n]$ needed to yield the missing tuples, the result being grouped and aggregated following A.

Intuitively, an instance-based explanation returns a set of modifications to the database, on which a grouping or aggregation constraint of A may apply. A modification $\mathcal{T}$ either corresponds to a labeled c-tuple C or again ϕ_{IB}, necessary for nested queries.

EXAMPLE 5. *The instance-based explanation of Ex. 1 is defined as follows, assuming that all ratings for P1 are above 2.*

$$\phi_{IB} = \{[\circ Product\langle \textsf{P1, Car, US}\rangle, -Ratings\langle \textsf{P1}, v_1, v_1 > 2\rangle, +Ratings\langle \textsf{P1}, v_2, v_2 <= 2\rangle]\}_{\textsf{P.ProdID = P1, P.Name = Car}}$$

In the rest of this paper, we will simplify the notation when possible, i.e., we will omit the subscript $\emptyset$ when no aggregation applies.

Let us now shift our attention to query-based explanations, returned for instance by Why-Not [4].

DEFINITION 6 (QUERY-BASED EXPLANATION). *A query-based explanation ϕ_{QB} for a debugging scenario is a set of query operators. Each operator $op_i \in \phi_{QB}$ is responsible for pruning missing-answers of E from $\mathcal{Q}(\mathcal{D})$ and satisfies $\mathcal{C}$. An operator op_i is responsible for pruning a missing answer $e \in E$ if data relevant to produce e is in the input of op_i but not in its output.*

The definition leaves open the choice of one or more operators in ϕ_{QB} and the choice of data relevant to produce e, as these are algorithm-specific.

EXAMPLE 6. *Given the query of Ex. 1 and the missing-answer $e = \langle$P1, Car, $v_1\rangle$, data relevant to produce e includes the tuple $\langle$P1, Car, US$\rangle \in Products$ and all tuples in $Ratings$. These data find "successors" in the output of all operators of the query, except for the selection $\sigma_{Max(R.Rating)\leq 2}$ (it is too selective). Consequently, $\phi_{QB} = \{\sigma_{Max(R.Rating)\leq 2}\}$.*

The final type of explanations to define before we define hybrid explanations are modification-based explanations.

DEFINITION 7 (MODIFICATION-BASED EXPLANATION). *A modification-based explanation ϕ_{MB} for a debugging scenario is a rewriting of $\mathcal{Q}$ into a set of queries $\mathcal{Q}'$ such that all missing tuples in E occur in $\mathcal{Q}'(D)$ for a given source instance D and $\mathcal{C}$ is satisfied.*

EXAMPLE 7. *A modification-based explanation for our example replaces the existing HAVING-clause by HAVING MAX(R.Rating) <= 5.*

As illustrated in Ex. 2, the above explanation types may fail in returning explanations in some scenarios. To extend the set of debugging scenarios for which explanations can be returned, we define a new type of explanation, namely hybrid explanation.

DEFINITION 8 (HYBRID EXPLANATION). *A hybrid explanation ϕ_H for a debugging scenario S is a 3-tuple $(\phi_{IB}, \phi_{QB}, \phi_{MB})$ s.t. the conjunction of all $\phi_i \in \phi_H$ is a valid explanation, even though any conjunction of a subset of ϕ_i's is not necessarily an explanation. A hybrid explanation is valid if, once data modifications of ϕ_{IB} were applied, ϕ_{QB} (ϕ_{MB}) would become valid query-based (modification-based) explanations w.r.t. S.*

EXAMPLE 8. *The hybrid explanation described in Ex. 3 is formally described as $(\phi_{IB}, \phi_{QB}, \perp)$ where*

$$\phi_{IB} = [\circ Products\langle \textsf{P3, Bus, CH}\rangle, +Ratings\langle \textsf{P3, 0}\rangle]$$
$$\phi_{QB} = \{\sigma_{P.Location=US}\}$$

4. THE *Conseil* ALGORITHM

We now describe *Conseil*, an algorithm implementing our framework by computing hybrid explanations of the form $(\Phi_{IB}, \Phi_{QB}, \perp)$. *Conseil* supports relational queries (i.e., queries involving selection σ, projection Π, join $\bowtie$, Cartesian product $\times$, union $\cup$, and set difference $\setminus$) with the restriction that it only supports one set difference. It also supports aggregation α. The rationale behind the restriction to one set difference operator per query is based on both complexity (see Sec. 4.3) and usability. Despite this restriction, we believe that *Conseil* is still widely applicable in practice.

In its current version, *Conseil* does not consider side-effects and we so far focus on explaining one missing-answer e to the result $Q(D)$ of a query Q over a relational instance D. However, *Conseil* exploits both referential constraints and unique constraints defined over D to estimate an explanation's cost. These are formalized in $\mathcal{C}$.

The above assumptions yield the following debugging scenario for *Conseil*: $S_{Conseil} = (\{e\}, \{Q\}, \{Q(D)\}, D, \mathcal{C})$.

Conseil operates in four main phases: First, (1) it computes a generic witness Φ that is then (2) annotated with passing properties determined over a canonical query tree representation (as defined in [6]). Based on the annotated generic witness, it (3) computes a set of derivations. Finally, (4) *Conseil* computes a hybrid explanation for each derivation, and returns these. We discuss each step in detail in the following. For illustration, we will use a more complex example than previously to cover more details.

EXAMPLE 9. *Fig. 2 shows the canonical query tree of a query Q over data in relations R, S, T, U, and V. Please ignore the rest of the figure for now. We define S_{Conseil} with $e = \langle$a', c', d'$\rangle$, Q and D as in Fig. 2, $Q(D) = \{\langle$a', c, d$\rangle\}$, and $C = \emptyset$.*

4.1 Step 1: Generic Witness Computation

First, *Conseil* compute a generic witness that describes a pattern each explanation conforms to and that comprises an instance-based component Φ_I and a query-based component Φ_Q. Essentially, we use this generic witness to limit the search space explored in subsequent steps. The generic witness can be computed efficiently based on e and Q (the complexity depending on the size of Q).

The instance-based component Φ_I describes in the form of c-tuples what data has to be present in the sources in order to produce e. Φ_Q, on the other hand, includes all operators that may be responsible for pruning e from the query result, i.e., σ, $\bowtie$, and $\setminus$.

To define the generic witness Φ, we first need to distinguish between missing-tuple constraints, subsequently called *mt-constraints*, and query-constraints, or *q-constraints* for short.

DEFINITION 9 (MT-CONSTRAINT). *An mt-constraint is a constraint that, given e and Q, can be identified as being imposed on the lineage [6] $D*$ of e w.r.t. Q by the missing-tuple e.*

DEFINITION 10 (Q-CONSTRAINT). *A q-constraint is a constraint that, given e and Q, can be identified as being imposed on the lineage $D*$ of e w.r.t Q by the query Q.*

EXAMPLE 10. *For Ex. 9, one mt-constraint is $R.A = a'$ whereas $U.B = u$ illustrates a q-constraint.*

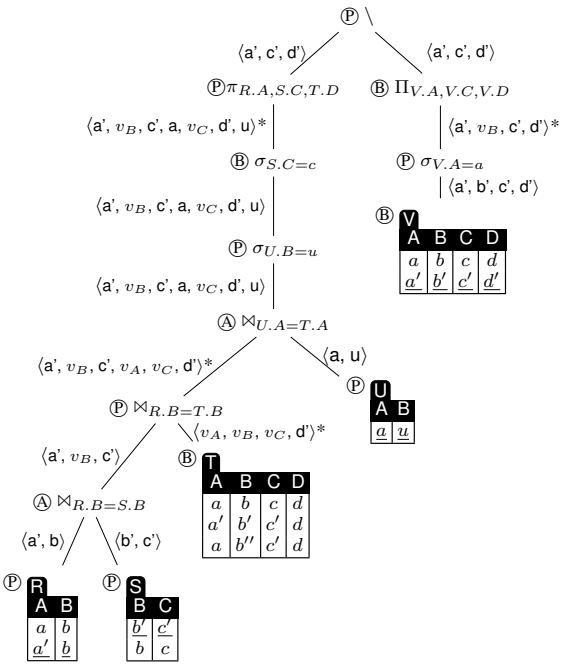

Figure 2: Sample query tree for scenario of Ex. 9

The only operators that introduce q-constraints are σ and $\bowtie$. Indeed, Π, α, and $\cup$ cannot be held responsible for pruning input tuples and $\setminus$ does not impose a constraint on the lineage of e, as lineage is defined solely by the tuples that need to be present.

We now define the generic witness for SPJD-queries, i.e., queries involving selection, projection, join, and set difference. We then comment on how to include union and aggregation.

DEFINITION 11 (GENERIC WITNESS Φ FOR SPJD-QUERIES). *A generic witness $\Phi = (\Phi_I, \Phi_Q)$ for a SPJD-query is a 2-tuple of sets Φ_I and Φ_Q, where $\Phi_I = \{t_i | relation(t_i) \in D* \text{ w.r.t. } e \text{ and } Q \wedge t_i.cond \text{ the mt-constraint on } t_i\}$ and $\Phi_Q = \{op_i | op_i \in Q \wedge op_i \in \{\sigma, \bowtie, \setminus\}\}$.*

EXAMPLE 11. *The generic witness for the debugging scenario of Ex. 9 has $\Phi_I = \{R\langle$a', ?$\rangle, S\langle$?,c'$\rangle, T\langle$?,?,?,d'$\rangle, U\langle$?,?$\rangle\}$[2] and $\Phi_Q = \{\bowtie_{R.B=S.B}, \bowtie_{R.B=T.B}, \bowtie_{U.A=T.A}, \sigma_{U.B=u}, \sigma_{S.C=c}, \setminus\}$.*

In case a query involves union operators, we create one generic witness for each alternative. For instance, $Q = (R \bowtie S) \cup T$ results in two generic witnesses, whose instance-based components have the form $\{R(\ldots), S(\ldots)\}$ and $\{T(\ldots)\}$, respectively.

When aggregation (together with grouping) is present, the c-tuples of the instance-based part of the generic witness are grouped accordingly, yielding a syntax for the generic witness similar to the syntax of the instance-based explanation, the main difference being that the c-tuples are not labeled and are sets instead of sequences (similar to the solution in [12]). For example, given a missing tuple ⟨b,3,c⟩ and a query $Q = \Pi_{B,A,C}(\alpha_{B,COUNT(A)ASA}(R) \bowtie S)$, we obtain $\Phi_I = \{\{R\langle v_a, \text{b}\rangle\}_{B,COUNT(A)=3}, S\langle$b, c$\rangle\}$.

4.2 Step 2: Generic Witness Annotation

Having the generic witness in hand, the next step is to annotate it with passing properties. We define three passing properties, named *passing*, *blocking*, and *ambiguous*. An operator is passing if we are certain that it is not responsible for pruning e from the result. If, on the contrary, we know that this operator is a culprit, we assign it the blocking annotation. In all other cases, we declare it as ambiguous.

[2]For conciseness, we use a Datalog inspired syntax to denote conditions, e.g., $R\langle r_1, r_2 \mid r_2 = s_1 \wedge r_1 = \text{a}\rangle, S\langle s_1, s_2 | s_1 = r_2\rangle \equiv R\langle$a$, r_2\rangle, S\langle r_2, ?\rangle$

Rule (1): $op_i = \sigma_{R_j.a\ cop\ c}$, where R_j is a table reference, a an attribute of table R_j, cop a comparison operator, and c an constant.	$\Phi'_I = (\Phi_i \setminus \{R_j(\ldots, a, \ldots \mid cond_j)\}) \cup R_j(\ldots, a, \ldots \mid cond_j \wedge (a\ cop\ c))$ $\Phi'_Q = \Phi_Q \setminus \{op_i\}$
Rule (2): $op_i = \bowtie_{R_j.a\ cop\ R_k.b}$ where a and b are attributes of tables R_j and R_k respectively and cop is a comparison operator.	$\Phi'_I = (\Phi_i \setminus \{R_j(\ldots, a, \ldots \mid cond_j), R_k(\ldots, b, \ldots \mid cond_k)\}) \cup R_j(\ldots, a, \ldots \mid cond_j \wedge (a\ cop\ R_k.b)) \cup T_k(\ldots, b, \ldots \mid cond_k \wedge (T_j.a\ cop\ b))$ $\Phi'_Q = \Phi_Q \setminus \{op_i\}$
Rule (3): $Q = Q_1 \setminus R_2$, where Q_1 is a subquery without difference and R_2 is a base relation. Furthermore, Q only contains one difference.	$\Phi'_I = \Phi_I \cup \{\not\exists R_2(getCTuple(\setminus, \Phi).a_1, \ldots, getCTuple(\setminus, \Phi).a_n, getCTuple(\setminus, \Phi).cond)\}$ $\Phi'_Q = \Phi_Q \setminus \{op_i\}$
Rule (4): $Q_1 \setminus Q_2$, where Q_1 and Q_2 are both queries without difference. Furthermore, Q only contains one difference.	let $v = Q_2(D)$, i.e., let v be the view defined by Q_2 over D. Then, we apply the same derivation rule as in the previous case (i.e., Rule (3)), with the difference that we have v instead of R_2.

Table 1: Derivation rules

DEFINITION 12 (ANNOTATED GENERIC WITNESS). *An annotated generic witness is a generic witness Φ where each c-tuple in Φ_I and each operator in Φ_Q is assigned an annotation. The set of possible annotations is the set $\{$Ⓐ, Ⓑ, Ⓟ$\}$, standing for ambiguous, blocking, and passing, respectively.*

Conseil first canonicalizes its input query Q into its canonical tree representation T. If query Q contains a set difference, we split the query into the left and the right subtree of the set difference. On the right subtree v, we call the function $annotate$ (described below) and, since we are in the negative part of the set difference, we revert passing annotations to blocking and vice versa. To correctly determine passing properties, we register the output of the subquery v over D (denoted $v(D)$) to V's topmost operator. We then process the left subtree, again calling $annotate$ and return the final annotated canonical tree T_A.

In discussing $annotate$, we invite the reader to follow the discussion on Fig. 2. We assume a function $getCTuple(R, \Phi)$ that extracts the c-tuple in the generic witness Φ that imposes constraints on relation R. We use this function to identify tuples in R compatible with the returned c-tuple. The compatible source tuples in our example are underlined in Fig. 2. We also show annotations. Essentially, a source relation is passing if at least one compatible tuple is found and blocking otherwise.

Compatible tuples are subsequently traced in a bottom-up fashion through T. In Fig. 2, edge labels show the trace of compatible tuples or, in case a compatible tuple is "lost" at a query operator, an "invented" c-tuple standing in as a place-holder for the lost compatible tuple (marked with*). Tracing such fictive tuples ensures that we can identify further operators responsible for pruning other compatible data. Based on this trace, a query operator is passing if every compatible input tuple has a corresponding compatible output tuple. On the other hand, it is blocking if the compatible input tuple violates a q-constraint of the operator. In all other cases, we assign the annotation ambiguous.

Tab. 2 summarizes the annotated generic witness, whose annotations equal the annotations of corresponding nodes of T_A.

4.3 Step 3: Derivation Computation

During derivation computation, *Conseil* refines the hybrid explanation pattern given by the generic witness by determining a *set* of patterns, called *derivations*.

Instance-based part	Query-based part
Annotated generic witness	
$\{R\langle$a',?$\rangle$Ⓟ, $S\langle$?,c'$\rangle$Ⓟ, $T\langle$?, ?, ?,d'$\rangle$Ⓑ, $U\langle$?, ?$\rangle$Ⓟ$\}$	$\{\bowtie_{R.B=S.B}$ Ⓐ, $\bowtie_{R.B=T.B}$ Ⓟ, $\bowtie_{U.A=T.A}$ Ⓐ, $\sigma_{U.B=u}$Ⓟ, $\sigma_{S.C=c}$Ⓑ, $\setminus$Ⓟ$\}$
Derivations	
$\{R\langle$a',$v_B\rangle$Ⓐ, $S\langle$?,c'$\rangle$Ⓟ, $T\langle$?,v_B,?,d'$\rangle$Ⓑ, $U\langle$?,u$\rangle$Ⓐ, $\not\exists v\langle$a',c',d'$\rangle$Ⓟ$\}$	$\{\bowtie_{R.B=S.B}$ Ⓐ, $\bowtie_{U.A=T.A}$ Ⓐ, $\sigma_{S.C=c}$Ⓑ$\}$
$\{R\langle$a',$v_B\rangle$Ⓐ, $S\langle v_B$,c'$\rangle$Ⓐ, $T\langle$?,v_B,?,d'$\rangle$Ⓑ, $U\langle$?,u$\rangle$Ⓐ, $\not\exists v\langle$a',c',d'$\rangle$Ⓟ$\}$	$\{\bowtie_{U.A=T.A}$ Ⓐ, $\sigma_{S.C=c}$Ⓑ$\}$
$\{R\langle$a',$v_B\rangle$Ⓐ, $S\langle$?,c'$\rangle$Ⓟ, $T\langle v_A,v_B$,?,d'$\rangle$Ⓑ, $U\langle v_A$,u$\rangle$Ⓐ, $\not\exists v\langle$a',c',d'$\rangle$Ⓟ$\}$	$\{\bowtie_{R.B=S.B}$ Ⓐ, $\sigma_{S.C=c}$Ⓑ$\}$
$\{R\langle$a',$v_B\rangle$Ⓐ, $S\langle v_B$,c'$\rangle$Ⓐ, $T\langle v_A,v_B$,?,d'$\rangle$Ⓑ, $U\langle v_A$,u$\rangle$Ⓐ, $\not\exists v\langle$a',c',d'$\rangle$Ⓟ$\}$	$\{\sigma_{S.C=c}$Ⓑ$\}$
Explanations	
$\{[\circ R\langle$a'b$\rangle, \circ S\langle$?, c'$\rangle, +T\langle$?, b, ?, d'$\rangle, \circ U\langle$a, u$\rangle]\}$	$\{\bowtie_{R.B=S.B}, \bowtie_{U.A=T.A}, \sigma_{S.C=c}\}$
$\{[\circ R\langle$a'b$\rangle, +S\langle$b, c'$\rangle, +T\langle$?, b, ?, d'$\rangle, \circ U\langle$a, u$\rangle]\}$	$\{\bowtie_{U.A=T.A}, \sigma_{S.C=c}\}$
$\{[\circ R\langle$a'b$\rangle, \circ S\langle$?, c'$\rangle, +T\langle$a, b, ?, d'$\rangle, \circ U\langle$a, u$\rangle]\}$	$\{\bowtie_{R.B=S.B}, \sigma_{S.C=c}\}$
$\{[\circ R\langle$a'b$\rangle, +S\langle$b, c'$\rangle, +T\langle$a, b, ?, d'$\rangle, \circ U\langle$a, u$\rangle]\}$	$\{\sigma_{S.C=c}\}$

Table 2: Generic witness, derivations, and explanations for scenario of Ex. 9

Given a generic witness $\Phi = (\{t_1, \dots t_n\}, \{op_1, \cdots, op_m\})$ for an SPJD-query, where $t_i = R_i(a_1, \dots, a_{n_i}, cond)$ and $op \in \{\sigma, \bowtie, \setminus\}$, we determine a derivation $\Phi' = (\Phi'_I, \Phi'_Q)$ by applying the derivation rules summarized in Tab. 1. In general, derivation is an iterative process that transfers one q-constraint of $op_i \in \Phi_Q$ into an mt-constraint in Φ_I.

We first apply the derivation rules to translate all passing operators of T_A into mt-constraints and corresponding c-tuples to be added to Φ_I. The intuition behind this is that a passing operator will never contribute to a the query-based component of a hybrid explanation. However, the conditions making it passing need to be satisfied by any instance-based component of a hybrid explanation. We apply the same idea to ambiguous operators next. However, as these operators stand for the possibility that the operator can be either passing or blocking, we create a derivation corresponding to each case. In general, assuming k is the number of ambiguous operators in T_A, there exist 2^k derivations. These derivations can be computed inductively in 2^k steps. Note that derivations also include annotations. These can be easily deduced during the derivation procedure based on a set of rules (omitted due to space constraints).

We can conceptually extend our derivation procedure to general relational queries involving more than one set difference operator. However, for *Conseil*, we exclude this case, because for generating actual explanations in the next step of the algorithm, we have to solve the view-update problem where updates are tuple deletions, for which solutions limit to conjunctive queries [3].

When dealing with queries involving union operators, we have seen that these will result in multiple generic witnesses, i.e., one for each subquery. In this case, we perform derivation for each produced generic witness. As for aggregation, we push conditions that apply to an aggregated result (e.g., $\sigma_{MAX(R.Rating \leq 2)}$) either into the c-tuples belonging to the grouped and aggregated sub-query (for MIN and MAX) or to agg itself (for COUNT, SUM, AVG). The reason for this differentiation lies in the fact that we do not actually want to update the source, and an explanation inserting or deleting a possibly large number of tuples just to match a certain count, sum, or average score is more difficult to interpret than just telling "there is a count, but it does not match your expectation".

EXAMPLE 12. *The derivations shown in Tab. 2 correspond to the derivation of all passing operators (line 1),* $\bowtie_{R.B=S.B}$ *(line 2),* $\bowtie_{R.B=T.B}$ *(line 3), and finally both joins (line 4).*

4.4 Step 4: Explanation Computation

In its final step, *Conseil* computes, for each derivation, a corresponding hybrid explanation with minimal cost. We first focus on determining the label assignment of the instance-based component of each derivation.

Given a derivation d, we preprocess it such that all unambiguous label assignments are determined beforehand. More specifically, we assign the label $\circ$ to all non-existential c-tuples (tuples not preceded by $\not\exists$ in the d) that are passing. On the other hand, if they are blocking, they are assigned the $+$-label. For the existential c-tuple (if any), we can remove it from the derivation's instance-based part if it is passing. If it is either ambiguous or blocking, we compute its lineage w.r.t. the view v. If the lineage is empty, it can be removed as well, otherwise, we assign it the $-$-label.

As a result of pre-processing, only ambiguous non-existential c-tuples remain to be further processed. The first step of this processing is to form clusters of relations for a given derivation d, where each cluster corresponds to the non-labeled relations of a connected component of the join graph of the instance-based component.

EXAMPLE 13. *For the last derivation in Tab. 2, preprocessing results in the partial c-tuple label assignment* $\{R\langle$,a',$v_B\rangle, S\langle v_B$, c'$\rangle$, $+T\langle v_A$, v_B, ?, d'$\rangle, U\langle v_A$, u$\rangle\}$ *and the clusters* $\{R, S\}$ *and* $\{U\}$.

For each cluster, we first establish a partial order of relations in a cluster based on a cost model (details omitted due to space constraints). More specifically, each relation X has one associated $maxCost(X)$ and $minCost(X)$. $MaxCost(X)$ quantifies the estimated worst case cost of modifying D in order to satisfy the constraints described by the c-tuple on X whereas $minCost(X)$ quantifies the cost of reusing existing data in D (that already satisfies the constraints). Based on this partial order, we span a binary search tree where a node N represents a relation and whose two edges to children have labels $\circ$ and $+$, respectively, standing for the two possible label-assignments for the c-tuple of the relation N represents. The root node corresponds to the relation with maximum $maxCost$ and its child nodes correspond to the next relation as determined by our order relation. The same applies for all subsequent levels. Using a branch-and-bound algorithm, we traverse this search-space and prune sub-trees if possible to eventually determine an instance-based component of a hybrid explanation with minimal cost.

The query-based component retains all query operators of the derivation's query-based component whose q-constraints are not satisfied by the determined instance-based component. The final minimal-cost hybrid explanations for our running example are summarized in Tab. 2.

5. EVALUATION

We implemented *Conseil*, Artemis [12], and Why-Not [4] in Java 1.6.. We ran all experiments on a Windows 7 installation running on a 1.7 GHz Intel Core i5 MacBook Air with 2 MB of main memory. We used a local installation of Postgres 9.2 as database system. As described in [4], lineage tracing relies on the Trio system(http://infolab.stanford.edu/trio/), which we also used in our implementation. For Artemis, we additionally uses Minion (available at http://minion.sourceforge.net).

We report results on an excerpt of our test queries, summarized in Tab. 3. We selected these queries as they are supported by all

Name	Expression
CRIME1	$\Pi_{p.name,c.type}(p \bowtie_{p.hair=sp.hair \wedge p.clothes=sp.clothes} sp \bowtie_{sp.witness=w.name} w \bowtie_{w.sector=c.sector} sector)$
CRIME 2	$\Pi_{p.name}(\sigma_{c.sector>97}(c) \bowtie_{c.sector=w.sector} w \bowtie_{w.name=sp.witness} sp \bowtie_{sp.hair=p.hair \wedge sp.clothes=p.clothes} p)$
MOV2	$\Pi_{loc.loc1}(\sigma_{l.year>1994}(l) \bowtie_{l.mid=rel.mid} rel \bowtie_{rel.lid=loc.lid} loc \bowtie_{l.title=r.title} \sigma_{r.rating\geq 9}(r))$
GOV1	$\Pi_{e.agency,e.bureau,e.title,e.desc,e.totalamount,es.substage}(e \bowtie_{e.eid=es.eid} es \bowtie_{es.sid=s.sid} (\sigma_{s.ln='Pelosi'}(s) \cup \sigma_{s.ln=X}(s))$
GOV3	$\Pi_{c.ln,c.fn,s.competed,s.name,s.state}(\sigma_{s.dollarsOblicated>10000}(s) \bowtie_{s.state=a.state.} a \bowtie_{a.id=c.id} c)$

Table 3: Queries used for evaluation

three algorithms. The queries originate from three different scenarios. The first scenario reuses the crime scenario used to evaluate Why-Not in [4]. The two other scenarios are based on real-world data from the movie and government domains [11]. To obtain competitive runtimes for Artemis, we added trust conditions on all but one table in all scenarios where necessary (i.e., all but MOV2).

Runtime comparison. From the results reported in Tab. 4, we see that both Why-Not and *Conseil* outperform Artemis and allow for interactive query debugging. The reason for this is that Artemis computes all possible instance-based explanations and needs to consider a large amount of alternatives. Opposed to that, both Why-Not and *Conseil* limit the result to the "best" explanations, providing a substantial advantage when considering runtime. Focusing on the relative performance of Why-Not and *Conseil*, we see that *Conseil* is slower than Why-Not in CRIME2, MOV2, and GOV3. Upon further analysis, we explain this based on the fact that in these cases, Why-Not stops very early in the process when the culprit operator is detected closely to the leaf nodes of the query tree, whereas *Conseil* performs more computations, as it also checks for possible culprit operators at higher levels by "inventing" c-tuples at the output of the first culprit operator. In CRIME1 and GOV1, *Conseil* is faster than Why-Not, as the just mentioned additional processing *Conseil* requires is compensated by the time Why-Not spends on computing the lineage of data in $Q(D)$ that is excluded from the data traced through the query.

Qualitative discussion. To briefly address the question of explanation quality, we report in Tab. 4 the number of explanations each algorithm returns. We observe that Artemis is not only slower than other algorithms, it also often produces too many instance-based explanations that may overwhelm the user. For CRIME1, we observe that Artemis returns no results, which is due to the fact that the crime to laugh is not present in the database, but it cannot be inserted by an instance-based explanation due to the trust condition on table c (the crime relation). This would also cause zero query-based explanations for Why-Not, if the first join of the canonical tree representation was a join involving this table. However, in our implementation, the join between p and sp comes first, which happens to also be a culprit operator (it filters the person named Roger). Opposed to that, *Conseil* returns two explanations. The first adds label $+$ to c-tuples on sp, w, and c, describing that both the crime c of laughing and a witness w that observed c (as described in table sp) are missing. The second explanation corresponds to a hybrid explanation that identifies both the missing crime being witnessed (i.e., $+c$ and $+w$) and the failing join between p and sp. Another interesting query is GOV3, where Why-Not does not return any result as necessary source data is missing, i.e., the state "CA" (which is "California" in the database). In all other cases, *Conseil* covers the query-based explanation of Why-Not as well as one (minimal-cost) instance-based explanation of Artemis.

6. CONCLUSION AND OUTLOOK

We presented *Conseil*, an algorithm that explains why data are missing from a query result using novel hybrid explanations. Opposed to previous work, *Conseil* also considers queries including set difference. We first set the theoretical foundation by providing a general framework to address the problem of explaining missing-answers. We then concentrated on defining *Conseil* to compute hybrid-explanations in four phases, namely generic witness computation, passing property annotation, derivation, and explanation generation. Experiments demonstrated that *Conseil* combines fast runtime with an explanation quality superior to explanations produced by other algorithms.

Next, we plan to consider side-effects and more general debugging scenarios. We also plan to further study efficiency improvements and cost models and to make a more thorough usability study.

Query	Missing-answer	Artemis	Why-Not	*Conseil*
CRIME1	⟨Roger, Laugh⟩	2.2s / 0	2.7s / 1	1.5s / 2
CRIME2	⟨Conedera⟩	11.9s / 424	0.5s / 1	1.8s / 2
MOV2	⟨Germany⟩	6.1s / 27	0.5s / 1	1.2s / 2
GOV1	⟨Edu, ?, ?, ?, v_4, Enacted, $v_4 \neq NULL$⟩	3.3s / 2	1.5s / 1	1.1s / 2
GOV3	⟨Pelosi, Nancy, ?, CA⟩	27.3s / 854	0.5s / 0	4.6s / 2

Table 4: Runtime (s) and #explanations for different algorithms

Acknolwedgements. Fundamental ideas of the *Conseil* algorithm have been developed in collaboration with Tim Belhomme. I also thank Hanno Eichelberger for his implementation work. Finally, I thank the Baden-Württemberg Stiftung for the financial support of this research project by the Eliteprogramme for Postdocs.

7. REFERENCES

[1] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems (TODS)*, 6(4):557–575, 1981.

[2] J. Bleiholder, S. Szott, M. Herschel, F. Kaufer, and F. Naumann. Subsumption and complementation as data fusion operators. In *International Conference on Extending Database Technology (EDBT)*, pages 513–524, 2010.

[3] P. Buneman, S. Khanna, and W. C. Tan. On propagation of deletions and annotations through views. In *Symposium on Principles of Database Systems (PODS)*, pages 150–158, 2002.

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

[5] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4), 2009.

[6] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179 – 227, 2000.

[7] J. Danaparamita and W. Gatterbauer. QueryViz: helping users understand SQL queries and their patterns. In *International Conference on Extending Database Technology (EDBT)*, pages 558–561, 2011.

[8] C. A. Galindo-Legaria. Outerjoins as disjunctions. In *International Conference on the Management of Data (SIGMOD)*, pages 348–358, 1994.

[9] T. Grust and J. Rittinger. Observing sql queries in their natural habitat (preprint). *ACM Transactions on Database Systems (TODS)*, 0(0), 2012.

[10] Z. He and E. Lo. Answering why-not questions on top-k queries. In *International Conference on Data Engineering (ICDE)*, pages 750–761, 2012.

[11] M. Herschel and H. Eichelberger. The Nautilus Analyzer: understanding and debugging data transformations. In *International Conference on Information and Knowledge Management (CIKM)*, pages 2731–2733, 2012.

[12] M. Herschel and M. A. Hernández. Explaining missing answers to SPJUA queries. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1):185–196, 2010.

[13] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1):736–747, 2008.

[14] T. Imieliński and J. Witold Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, 1984.

[15] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. SnipSuggest: Context-aware autocompletion for SQL. *Proceedings of the VLDB Endowment (PVLDB)*, 4(1):22–33, 2010.

[16] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proceedings of the VLDB (PVLDB)*, 4(1):34 – 45, 2010.

[17] A. Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *Proceedings of the VLDB (PVLDB)*, 4(12):1466–1469, 2011.

[18] Q. T. Tran and C.-Y. Chan. How to ConQueR why-not questions. In *International Conference on the Management of Data (SIGMOD)*, pages 15 – 26, 2010.