# 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-monotonous queries. Experiments on efficiency and explanation quality show that *Conseil* is comparable and even outperforms previous algorithms.

## 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.

The problem of simplifying the understanding, analysis, and debugging of complex data transformations, and in particular SQL and relational queries, has led to the development of a variety of techniques [5, 7, 9, 12]. 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). Further use-cases of finding missing-answers include what-if analysis for query behavior or the generation of queries for benchmarking purposes, where generated queries ideally do not return a non-empty result.

Recently, approaches to explain missing-answers of relational and SQL queries have been proposed. Essentially, these approaches generate either *instance-based explanations* [13, 14], *query-based explanations* [4], or *modification-based explanations* [19]. The following example, used and extended throughout the paper, illustrates the different types of explanations.

EXAMPLE 1. *Figure 1 shows an SQL query and sample input data. The query determines which products obtained low ratings in the US, a query useful for instance to decide which products should be discontinued. We assume ProdID is a primary key in Products.*

```
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**

*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 [13, 14][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, 19] 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

[1] [14] also considers updating attribute values of existing tuples in instance-based explanations. However, it is easy to construct a scenario where even this approach fails in delivering a result, e.g., if we set the attribute as not updatable, an option suggested in [14].

.

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*.

EXAMPLE 3. *In the scenario of Example 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-monotonous** queries. We further define **relationships** between all different types of explanations. We also present the ***Conseil* algorithm** that computes hybrid explanations. An additional and substantial novelty compared to other algorithms is its support of **non-monotonous queries**. We **experimentally compare *Conseil* to existing algorithms**, focusing both on **runtime** and on the **quality of explanations** returned.

In the rest of this paper, we first analyze related work in Section 2. Next, we formalize our framework to compute hybrid explanations in Section 3. Section 4 focuses on the *Conseil* algorithm, which is evaluated in Section 5 before we conclude in Section 6.

# 2. RELATED WORK

As mentioned previously, 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, including data lineage [6] and more generally data provenance [5], sub-query result inspection [9], visualization [7], or transformation specification simplification [11, 12, 16, 18]. The work presented in this paper falls in the category of data provenance research, focusing on a specific subproblem referred to as why-not provenance [4, 17]. Due to the lack of space, we focus the remainder of the discussion of related work to existing approaches addressing the why-not provenance issue by generating different kinds of explanations to missing-answers.

The Missing-Answers (MA) algorithm [14] 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 explanation 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 [13] 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. The computed instance-based explanations describe all possible explanations that insert source data such that the simultaneous lack of the set of specified missing-answers can be explained. Artemis also considers so called 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 [19], 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 (i.e., upon query modification, tuples existing in the original query result must remain and only a minimal number of additional tuples are admissible).

Another algorithm that computes modification-based explanations while considering side-effects specializes on answering why-not questions on top-k queries [10]. Here, the 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 (part of future work) nor top-k queries.

# 3. FRAMEWORK AND DEFINITIONS

To set the theoretical foundation of *Conseil*, we first extend the definitions of the different explanation types to fit the most general scenario. Similarly to [13], we call these *debugging scenarios*. We then define hybrid explanations and show interesting relationships between different types of explanations.

## 3.1 Framework

We define a debugging scenario for the general case where multiple missing-answers and multiple queries are considered. These definitions capture all previous definitions and offer enough freedom to allow for further algorithms.

To define a debugging scenario, we first have to define conditional tuples [15] as well as matching conditional tuples.

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 belongs to, we use $R\langle a_1, \ldots, a_n, cond\rangle$. Also, we refer to an attribute $a$ within a c-tuple $t$ using the notation $t.a$, e.g., $t.cond$ refers to the condition of the c-tuple.

We now define a debugging scenario, which represents the input of an algorithm explaining missing-answers.

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 illustrate how the algorithms surveyed in Section 2 conform to this framework. We can describe the debugging scenario of MA as $(\{e\}, \{Q\}, \{Q(D)\}, D, \mathcal{C})$. More specifically, MA is designed to explain one missing tuple $e$ from the result of one query $Q$. The constraints are the trust constraints. Artemis defines a debugging scenario $S = \{E, \mathcal{Q}, \mathcal{Q}(\mathcal{D}), D, \mathcal{Q}_m, \mathcal{Q}_{im}\}$ where $\mathcal{Q}_m$ and $\mathcal{Q}_{im}$ describe constraints that minimize or prohibit side-effects on designated query results, respectively. As a consequence, these can be seen as constraints, i.e., to conform to our framework, we can set $\mathcal{C} = \{\mathcal{Q}_m, \mathcal{Q}_{im}\}$. As for Why-Not, it takes a predicate in disjunctive normal form as input that can be interpreted as a set of c-tuples describing missing-answers to a single query $Q$. No further constraints apply, so the debugging scenario for Why-Not corresponds to $\{E, \{Q\}, \{Q(D)\}, D, \emptyset\}$. Finally, ConQueR explains multiple missing-answers from one query $Q$ and allows to specify constraints spanning multiple missing-answers, which can be modeled by $\mathcal{C}$. That is, we have $(E, \{Q\}, \{Q(D)\}, D, \mathcal{C})$ as debugging scenario for ConQueR.

A debugging scenario defines the input provided to an algorithm that computes explanations for missing-answer. Let us now turn to the definition of its output. As mentioned previously, different algorithms return different types of explanations. We define these for the case where $\mathcal{Q}$ consists of queries involving operators from relational algebra plus aggregation. Thus, below definitions extend previous definitions of query-based, instance-based, and modification-based explanations, that so far have been defined for a fragment of relational queries (with aggregation).

An instance-based explanation generally consists of labeled c-tuples. The definition of these relies on the concept of compatible c-tuples, defined first.

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 $\phi_{IB}$ 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 $\phi_{IB}$ is:*

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

*where $L\langle a_1, \ldots, a_n$, cond$\rangle$ refers to Definition 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 $\phi_{IB}$ 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 $\phi_{IB}$, necessary for nested queries.

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

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

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 $\phi_{QB}$ 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 $\phi_{QB}$ as this depends on the properties and optimizations an algorithm implements. For instance, Why-Not returns a set of operators closest to the root in the canonical query plan of $\mathcal{Q}$ (the canonical representation being defined in [6]) that prune any missing-answer. This definition also leaves open the choice of data relevant to produce $e$, as this is also algorithm-dependent. Why-Not for instance chooses to select tuples in the source database containing attributes compatible with at least one attribute in the c-tuple defining $e$ and that are not in the lineage of any result tuple in $Q(D)$.

EXAMPLE 6. *Given the query of Example 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 $Ratings$. Joining both tables on $PID$, a tuple consisting of "successors" of relevant tuples, i.e., $\langle$P1, Car, US, P1, 5$\rangle$ occurs in the output of the join. This result tuple satisfies $\sigma_{P.loc=US}$ and thus finds a successor in the selection's output. The same is true for the aggregation operator, retuning $\langle$P1, Car, 5$\rangle$. However, this tuple does not pass the subsequent selection $\sigma_{Max(R.Rating)\leq 2}$. Consequently, this selection operator is identified as culprit operator for e and $\phi_{QB} = \{\sigma_{Max(R.Rating)\leq 2}\}$.*

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 $\phi_{MB}$ 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 running example may rewrite the original SQL query as follows:*

```
SELECT P.ProdID, P.ProdName
FROM ... WHERE ... GROUP BY ...
HAVING MAX(R.Rating) <= 5
```

As illustrated in Example 2, the above explanation types may fail in returning explanations in some scenarios. In some cases, one explanation type may compensate for the empty result of another explanation type, but we have seen that it is possible that no explanation type can generate explanations on its own. 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 $\phi_H$ 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 $\phi_i$'s is not necessarily an explanation. A hybrid explanation is valid if, once data modifications of $\phi_{IB}$ were applied, $\phi_{QB}$ ($\phi_{MB}$) would become valid query-based (modification-based) explanations w.r.t. $S$.*

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

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

## 3.2 Explanation Type Relationships

Having defined the different explanation types, we briefly provide interesting relationships that hold between these explanation types. Proofs are trivial and are omitted due to space constraints.

THEOREM 1. *Given an explanation $\phi_i$, where $i \in \{IB, QB, MB\}$ it is true that $\phi_i$ also qualifies as hybrid explanation. The converse, however, is not true. We write $\phi_i \to \phi_H, i \in \{IB, QB, MB\}$.*

THEOREM 2. *Let $\Phi_i$ be the set of* all possible *explanations of type $i \in \{IB, QB, MB, H\}$. Then, $\Phi_{IB} \cup \Phi_{QB} \cup \Phi_{MB} \subseteq \Phi_H$.*

THEOREM 3. *If there exists a query-based explanation, there also exists an equivalent modification-based explanation. The converse is also true. Thus, the set of all query-based explanations covers the same cases as the set of all modification-based explanations, i.e., $\Phi_{QB} \equiv \Phi_{MB}$.*

THEOREM 4. *We can simplify the definition of a hybrid explanation to one of the two following definitions without information loss: $\phi_{H1} = \{\phi_{IB}, \phi_{QB}\}$ and $\phi_{H2} = \{\phi_{IB}, \phi_{MB}\}$.*

Note that the above theorems cover the general theoretical case. As different algorithms target different subsets of the general problem defined by our framework, these may not hold. For instance, there is no equivalence between the query-based explanations Why-Not returns and the modification-based explanations ConQueR computes. For instance, ConQueR solely deals with numerical data, whereas Why-Not also considers string data.

# 4. THE *Conseil* ALGORITHM

Having described the general framework for explaining missing-answers, we now describe *Conseil*, an algorithm implementing our framework by computing hybrid explanations and that, consequently, also covers all other types of explanations.

*Conseil* computes hybrid explanations for relational queries (i.e., queries involving selection $\sigma$, projection $\Pi$, join $\bowtie$, Cartesian product $\times$, union $\cup$, and set difference $\setminus$) with the restriction that it only supports one set difference operator. It also supports aggregation $\alpha$. The rationale behind the restriction to one set difference operator per query is twofold. First, as we shall see, having set difference operators in our query will bring us to solving a specific type of view deletion problem, and by restricting ourselves to one set difference operator, we can leverage existing approaches to solving this problem. Second, generating hybrid explanations for queries with more than one difference is not very practical as the resulting explanation (if it can be computed at all) easily becomes too difficult for the developer to interpret. 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 $D$, which are formalized in $\mathcal{C}$.

**Algorithm 1:** *Conseil* Algorithm

```
Input: a debugging scenario ({e}, {Q}, {Q(D)}, D, C)
Output: set of hybrid explanations φ_H
Φ ← computeGenericWitness(e, Q);
T_A ← annotatePassingProperties(Q, D, Φ);
𝒟 ← computeDerivations(Φ, T_A);
φ_H ← ∅;
foreach derivation d ∈ 𝒟 do
    φ_H ← φ_H ∪ computeExplanations(d, D);
return φ_H;
```

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

Algorithm 1 highlights the four main steps of *Conseil*. First, it computes a generic witness $\Phi$ that is then annotated with passing properties. Based on the annotated generic witness $T_A$, it computes a set of derivations $\mathcal{D}$. Finally, *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. *Figure 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 $S_{\text{Conseil}}$ as follows[2]:*

$$\begin{aligned} e &= \langle \textit{a', c', d'} \rangle \\ Q, D &= \textit{see Figure 2} \\ Q(D) &= \{\langle \textit{a', c, d} \rangle\} \\ C &= \{\underline{R.A}, \underline{S.B}, \underline{T.B}, \underline{U.A}, R.B \to S, R.B \to T, T.A \to U\} \end{aligned}$$

## 4.1 Step 1: Generic Witness Computation

First, *Conseil* compute a generic witness. Intuitively, a generic witness describes a pattern each explanation conforms to. Similarly to a hybrid explanation, a generic witness $\Phi$ comprises an instance-based component, denoted $\Phi_I$, and a query-based component, denoted $\Phi_Q$. 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 $\Phi_I$ describes in the form of using c-tuples what data has to be present in the sources in order to produce $e$. $\Phi_Q$, on the other hand, includes all operators that may be responsible for pruning $e$ from the query result, i.e., $\sigma$, $\bowtie$, and $\setminus$.

To compute the generic witness, we extend the definition of a generic witness for instance-based explanations [13], defined for a query that, in terms of [6] corresponds to a single $\alpha$-$\cup$-$\Pi$-$\sigma$-$\bowtie$-segment (i.e., an SPJUA query). Our extension allows to have a query with an arbitrary number of these segments, either connected directly to each other or, in just one instance, connected through one additional set difference operator (i.e., a single $\setminus$-segment).

To define the generic witness $\Phi$, 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][3] $D*$ of $e$ w.r.t. $Q$ by the missing-tuple $e$.*

[2] $\underline{R.X}$ identifies attribute $X$ as key attribute in $R$ and $R.X \to T$ describes a foreign key $X$ in relation $R$ referencing relation $T$.

[3] In [6], this actually corresponds to their definition of derivation, however, we use the term lineage here as we use the term derivation in a different context.

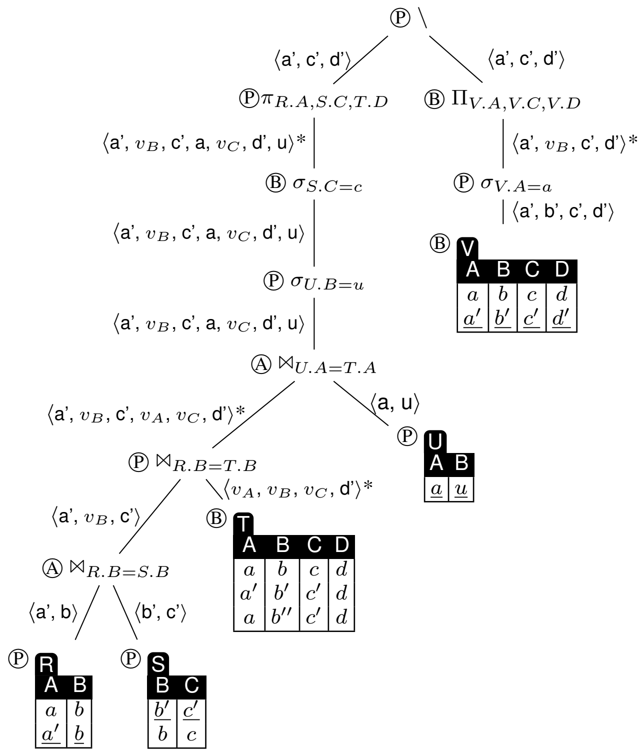


**Figure 2: Sample query tree for scenario of Example 9**

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

The only operators that introduce q-constraints are $\sigma$ and $\bowtie$. Indeed, $\Pi, \alpha$, 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. Note that whenever the detailed discussion of *Conseil* focuses on this query fragment, we make comments on how to include union and aggregation.

DEFINITION 11 (GENERIC WITNESS $\Phi$ FOR SPJD-QUERIES). *A generic witness $\Phi = (\Phi_I, \Phi_Q)$ for a SPJD-query is a 2-tuple of sets $\Phi_I$ and $\Phi_Q$, where $\Phi_I$ is the instance-based component of $\Phi$ and $\Phi_Q$ is its query-based component. $\Phi_I$ includes a c-tuple for each relation in the lineage of* e *w.r.t. Q, c-tuple conditions corresponding to mt-constraints on the respective tables. More formally,*

$$\Phi_I = \{t_i | relation(t_i) \in lineage(e, Q) \wedge t_i.cond \text{ the mt-constraint on } t_i\}$$

*$\Phi_Q$ contains all query operators of Q that result in a q-constraint plus the set-difference operator (if present), i.e.,*

$$\Phi_Q = \{op_i | op_i \in Q \wedge op_i \in \{\sigma, \bowtie, \setminus\}\}$$

EXAMPLE 11. *The generic witness for the debugging scenario of Example 9 has $\Phi_I = \{R\langle \text{a'}, ?\rangle, S\langle ?, \text{c'}\rangle, T\langle ?, ?, ?, \text{d'}\rangle, U\langle ?, ?\rangle\}$[4] and $\Phi_Q = \{\bowtie_{R.B=S.B}, \bowtie_{R.B=T.B}, \bowtie_{U.A=T.A}, \sigma_{U.B=u}, \sigma_{S.C=c}, \setminus\}$. This generic witness describes the fact that in order for* e *to become part of $Q(D)$, we need some tuple in* R *that contributes the value* a'*, a tuple in* S *to satisfy* S.c = c'*, a tuple in* T *with a value* d' *and a tuple in* U*. The query-based component includes all join, selection, and set difference operators of the sample query.*

[4] For conciseness, we use a syntax similar to the notation used in the body of Datalog rules to denote conditions, e.g., $R\langle r_1, r_2 \mid r_2 = s_1 \wedge r_1 = \text{a}\rangle, S\langle s_1, s_2 | s_1 = r_2\rangle \equiv R\langle \text{a}, r_2\rangle, S\langle r_2, ?\rangle$

**Algorithm 2:** $annotatePassingProperties(Q, D, \Phi)$

**Input**: query $Q$, source data $D$, generic witness $\Phi$
**Output**: $T_A$, the canonical query tree annotated with passing properties

```
T ← canonicalize(Q);
T'_A ← T;
V ← right subtree of single set difference operator in T;
if V ≠ null then
    T_{A,V} ←
    annotate(V, D, genericWitness(getCtuple(\, Φ), V));
    foreach node n ∈ T_{A,V} do
        if n.pp = Ⓟ then
            n.pp ← Ⓑ;
        else if n.pp = Ⓑ then
            n.pp ← Ⓟ ;
    V.root.output = V(D);
    T'_A ← T after replacing subtree V by T_{A,V};
T_A ← annotate(T'_A, D, Φ);
Φ ← getAnnotationsFromTree(T_A, Φ) ;
return T_A;
```

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. For example, given a missing tuple $\langle \text{b,3,c}\rangle$ and a query $Q = \Pi_{B,A,C}(\alpha_{B,COUNT(A)ASA}(R) \bowtie S)$, we obtain $\Phi_I = \{\{R\langle v_a, \text{b}\rangle\}_{B,COUNT(A)=3}, S\langle \text{b, c}\rangle\}$.

## 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.

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

The annotation step allows us to refine the explanation pattern given by the generic witness. Indeed, after determining the passing properties of query operators (notably those included in $\Phi_Q$), we will "discard" passing operators (as they cannot be held responsible for making the missing-answer disappear) and "fix" blocking operators, meaning that these will be part of the query-based component of any hybrid explanation. Consequently, *Conseil* subsequently focuses on "resolving" the ambiguity of the remaining operators.

Algorithm 2 shows the annotation procedure for any possible input query to *Conseil*. It first canonicalizes its input query $Q$ into

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

**Table 1: Generic witness, derivations, and explanations for scenario of Example 12**

**Algorithm 3:** $annotate(T, D, \Phi)$

```
   Input: canonical query tree T, source data D, generic witness Φ
   Output: T_A, the canonical query tree annotated with passing properties
1  Q ← serialize(T) ;
2  foreach relation type r ∈ Q do
3      if r contains tuples compatible with getCTuple(r, Φ) then
4          r.pp ← Ⓟ;
5          r.output ← all tuples from r in D;
6      else
7          r.pp ← Ⓑ;
8          r.output ← getCTuple(r, Φ) ;
9  foreach op ∈ Q do
10     foreach c ∈ op.children do
11         op.input ← op.input ∪ {(c, c.output} ;
12     op.output ← op.run();
13     if op.output contains a "relevant successor" then
14         op.pp ← Ⓟ;
15     else if op.cond is not compatible with at least one mt-constraint in Φ then
16         op.pp ← Ⓑ;
17         op.output ← getCTuple(op, Φ);
18     else
19         op.pp ← Ⓐ;
20         op.output ← getCTuple(op, Φ) ;
21 T_A ← buildTree(Q);
22 return T_A;
```

its canonical tree representation $T$[5]. 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 $annotate$ (discussed 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 result $T_A$.

Algorithm 3 describes the annotation assignment procedure that applies on any sequence of $\alpha$-$\cup$-$\Pi$-$\sigma$-$\bowtie$-segments. It first serializes the canonical tree $T$ into a queue $Q$. This serialization makes sure that all children nodes precede their parent node. We then traverse the first elements of the queue that are relation types in lines 2 through 8, before we traverse query operators in lines 9 through 20. Due to the lack of space, we do not further discuss the internals of the algorithm. To provide a general understanding, we illustrate its functionality on our complex example. We assume that $getCTuple(r, \Phi)$ extracts the c-tuple in the generic witness $\Phi$ that imposes constraints on relation $r$.

EXAMPLE 12. *The query of Figure 2 includes a set difference, hence, its right subtree will be annotated first. The c-tuple returned by* $getCTuple(\setminus, \Phi)$ *is* ⟨a',c',d'⟩. *The generic witness on this subtree includes the instance-based component*$\{V\langle \text{a'},v_B, \text{c'}, \text{d'}\rangle\}$, *for which we find the underlined compatible tuple in* $V$. *Hence, the annotation as determined by Algorithm 3 is* Ⓟ, *which is then changed into* Ⓑ *by Algorithm 2. We register the compatible source tuple of* $V$ *to the output of relation schema* $V$ *and proceed to* $\sigma_{V.A=a}$. *This operator is identified as blocking (later converted to passing) as its condition contradicts the mt-constraint on* $V$ *that requires* $V.a = a'$. *As a consequence, we streamline an "invented" tuple* ⟨a',$v_B$, c', d'⟩ *to the projection, a generally passing (and in the right sub-tree of a set difference thus blocking) operator. Next, we pass on the result of the subquery as right input to the set difference.*

*Focusing on the left subtree of the set difference, we find compatible source data in both R and S, which are thus marked passing and that pass on their compatible output to their parent. Because T does not contain any tuple where D = d', it is blocking and passes on a c-tuple* $\langle v_A, v_B, v_C, \text{d'}\rangle$. *The join between tables R and S is ambiguous because no tuple in its result combines the values a' and c'. Consequently, a c-tuple encoding the join condition as a common unknown value* $v_B$ *is registered to the subsequent join. This join, i.e.,* $\bowtie_{R.B=T.B}$ *is passing because, given its input, which consists of c-tuples, its output contains a tuple that is compatible with the desired missing-answer. This compatible output c-tuple is registered as input to the final join* $\bowtie_{U.A=T.A}$. *This join results in no compatible result c-tuple, so it is ambiguous and we invent a compatible tuple for the subsequent operators. For these, it is easy to verify that the c-tuple* ⟨a', $v_B$, $v_C$, d', a', $v_U$⟩ *has compatible successors for both selections and the projection, so all these operators are passing. Finally, we observe that the set difference is passing, because no compatible left input is eliminated by its right input.*

*Table 1 summarizes the annotated generic witness, whose annotations are simply equal to the annotations of corresponding nodes of* $T_A$, *as assigned in line 14 of Algorithm 2.*

## 4.3 Step 3: Derivation Computation

As mentioned before, we annotate the generic witness to refine the hybrid explanation pattern given by the generic witness. More specifically, we determine a set of patterns, called *derivations*.

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 Table 2. In general, derivation is an iterative process that transfers one q-constraint of $op_i \in \Phi_Q$ into an mt-constraint in $\Phi_I$.

Algorithm 4 describes the derivation procedure, It first applies the derivation rules to translate all passing operators of $T_A$ into mt-constraints and corresponding c-tuples to be added to $\Phi_I$. The intuition behind this is that a passing operator will never contribute to a query-based explanation (or the query-based part of a hybrid explanation). However, the conditions making it passing need to be satisfied by any instance-based explanation (or instance-based part of a hybrid explanation). We apply the same idea to ambiguous

[5]We determine a canonical tree representation as defined in [6].

| | |
|---|---|
| **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. | $\Phi'_I = (\Phi_i \setminus \{R_j(\ldots, a, \ldots \mid cond_j)\})$<br>$\cup R_j(\ldots, a, \ldots \mid cond_j \wedge (a\ cop\ c))$<br>$\Phi'_Q = \Phi_Q \setminus \{op_i\}$ |
| **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. | $\Phi'_I = (\Phi_i \setminus \{R_j(\ldots, a, \ldots \mid cond_j),$<br>$R_k(\ldots, b, \ldots \mid cond_k)\})$<br>$\cup R_j(\ldots, a, \ldots \mid cond_j \wedge (a\ cop\ R_k.b))$<br>$\cup T_k(\ldots, b, \ldots \mid cond_k \wedge (T_j.a\ cop\ b))$<br>$\Phi'_Q = \Phi_Q \setminus \{op_i\}$ |
| **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. | $\Phi'_I = \Phi_I \cup$<br>$\{\nexists R_2(getCTuple(\setminus, \Phi).a_1,$<br>$\ldots,$<br>$getCTuple(\setminus, \Phi).a_n,$<br>$getCTuple(\setminus, \Phi).cond)\}$<br>$\Phi'_Q = \Phi_Q \setminus \{op_i\}$ |
| **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 2: Derivation rules**

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 (see Algorithm 5).

Essentially, we obtain a derivation by applying a sequence of derivation rules to the generic witness $\Phi$. We denote $\langle a, b\rangle$ the derivation sequence that first applies derivation rule $a$ and then derivation rule $b$. The resulting witness is denoted as $\Phi_{\langle a,b\rangle}$. It is easy to show that $\Phi_{\langle a,b\rangle} \equiv \Phi_{\langle b,a\rangle}$, a fact we exploit to reduce the number of derivation sequences to explore. In general, assuming $k$ is the number of ambiguous operators in $T_A$, there exist $2^k$ derivations. These derivations can be computed inductively, meaning that these $2^k$ derivations can be computed in $2^k$ steps.

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 [1], restricting to the case where the update is a deletion. For this view-deletion problem, we ultimately plan to leverage previous results [3] obtained for conjunctive queries to be efficient and to minimize side-effects. In our current implementation, we however produce explanations that delete the lineage of any tuple in $v$ matching the c-tuple $getCTuple(\setminus, \Phi)$.

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".

So far, we have discussed the derivation without considering passing properties. However, just like a generic witness, each derivation has passing properties, determined as follows.

- If $op \in \{\sigma, \bowtie\}$ has only passing descendants, and $op$ is passing, then the annotation of the modified c-tuples is Ⓟ.
- A blocking c-tuple remains blocking after derivation.
- The passing property of the *existential c-tuple* in $\Phi'_I$ (the c-tuple preceded by $\nexists$ in Rule (3)) introduced by the set difference operator inherits the operator's annotation in $\Phi_Q$.
- In all other cases, the passing property is set to Ⓐ.
- $\Phi'_Q$ retains the passing properties of $\Phi_Q$.

EXAMPLE 13. *The derivations shown in Table 1 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, the corresponding set of hybrid explanations. At this stage of the algorithm, the query-based component of an explanation is given by the set of operators in the query-based component of a derivation. As a consequence, this step focuses on exploring the possible label assignments in the instance-based component of each derivation.

In principle, *Conseil* could use any algorithm to compute instance-based explanations (and limiting the output to those conforming to the derivation's "pattern"). However, existing algorithms [13, 14] compute *all* instance-based explanations, their number increasing exponentially with the data. This is both time consuming and the result may be too overwhelming for the developer to be of any use. Hence, *Conseil* limits to the computation of the "cheapest" hybrid explanation for each derivation, based on a cost model.

**Efficiently computing hybrid explanations.** Algorithm 6 describes the general explanation generation procedure. Given a derivation $d$, we preprocess it such that all unambiguous label assignments are determined beforehand. More specifically, the $preprocess(d)$ assigns the label $\circ$ to all non-existential c-tuples 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.

**Algorithm 4:** $computeDerivations(\Phi, T_A)$

**Input**: generic witness $\Phi$, annotated canonical query tree $T_A$
**Output**: $\mathcal{D}$, the set of derivations of $\Phi$ w.r.t. $T_A$
$\mathcal{D} \leftarrow \emptyset$ set of derivations, initially empty;
$\Phi' \leftarrow \{\Phi'_I, \Phi'_Q\}$, a derivation with initially empty $\Phi'_I$ and $\Phi'_Q$;
$A \leftarrow$ set of ambiguous operators, initially empty;
**foreach** $op \in \Phi_Q$ **do**
    **if** $op.pp =$ Ⓟ **then**
        $\Phi' = applyDerivationRules(op, \Phi)$;
    **else if** $op.pp =$ Ⓐ **then**
        $A \leftarrow A \cup \{op\}$;
$\mathcal{D} \leftarrow findDerivationsForAmbiguous(A, \Phi')$;
**return** $\mathcal{D}$;

**Algorithm 5:** $findDerivationsForAmbiguous(A, \Phi)$

**Input**: set of ambiguous operators $A$, a derivation $\Phi$
**Output**: $\mathcal{D}$, a set of derivations based on $\Phi$
$\mathcal{D} \leftarrow \emptyset$;
$op \leftarrow A[0]$;
$\Phi'_1 \leftarrow applyDerivationRules(op, \Phi)$;
$\Phi'_2 \leftarrow (\Phi_I, A)$;
$\mathcal{D} \leftarrow \{\Phi'_1, \Phi'_2\}$;
$\mathcal{D} \leftarrow \mathcal{D} \cup findDerivationsForAmbiguous(A \setminus \{op\}, \Phi'_1) \cup findDerivationsForAmbiguous(A \setminus \{op\}, \Phi'_2)$;
**return** $\mathcal{D}$;

EXAMPLE 14. *For the last derivation in Table 1, preprocessing results in the partial c-tuple label assignment* $\{R\langle, \mathsf{a'}, v_B\rangle, S\langle v_B, \mathsf{c'}\rangle, +T\langle v_A, v_B, ?, \mathsf{d'}\rangle, U\langle v_A, \mathsf{u}\rangle\}$ *and the clusters* $\{R, S\}$ *and* $\{U\}$.

When clusters contain only one relation X, we can easily conclude that generated explanations with minimal cost will reuse existing tuples from X that satisfy the conditions on X, if any exist. For instance, for cluster $\{U\}$ in Example 14, we find $\langle \mathsf{a}, \mathsf{u}\rangle \in U$ that satisfies the constraints of the c-tuple $U\langle v_A, \mathsf{u}\rangle$. As we can reuse existing data for $U$, we assign the $\circ$-label to its associated c-tuple.

Processing clusters containing more than one relation is more challenging. Based on a cost model (described below), we first establish a partial order of relations in a cluster. 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).We then call $descendExplanationTree$, which spans a binary search tree as discussed below. Due to space constraints, we omit the pseudocode for $descendExplanationTree$ and limit here to a detailed discussion of $descendExplanationTree$. Note that this algorithm computes the instance-based component of an explanation. 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 (line 9). The final minimal-cost hybrid explanations for our running example are summarized in Table 1.

EXAMPLE 15. *To illustrate all steps of the algorithm, assume that the c-tuple over* T *is ambiguous, e.g., because* $\langle \mathsf{a}, \mathsf{b}, \mathsf{c'}, \mathsf{d'}\rangle \in T$. *Hence, for the fourth derivation, we obtain one cluster* $\mathcal{C} = \{R, S, T, U\}$. *Let us assume the following partial order relation:*

$$\begin{array}{c} maxCost(R) \geq maxCost(S) \geq maxCost(T) \geq maxCost(U) \\ minCost(R) = minCost(S) = minCost(T) = minCost(U) \end{array} \quad (1)$$

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 a hybrid explanation with minimal cost.

EXAMPLE 16. *Figure 3 illustrates the tree describing all possible combinations of assigning labels for the cluster of Example 15. Dotted edges represent the paths pruned by our algorithm. The figure also shows which subtrees were pruned during intermediate pruning steps, labelled steps (1) through (6).*

**Algorithm 6:** $computeExplanations(d, D)$

**Input**: derivation $d$, source data D
**Output**: set of hybrid explanations for derivation $d$, denoted $\phi_H^d$

1 $\phi_H^d \leftarrow \{\phi_{IB}^d, \phi_{QB}^d\}$, with $\phi_{IB}^d$ and $\phi_{QB}^d$ initially empty;
2 $d \leftarrow preprocess(d)$;
3 $\mathcal{C} \leftarrow identifyClusters(d)$;
4 **foreach** *cluster* $C \in \mathcal{C}$ **do**
5 $\quad C_O \leftarrow sortRelationsByCost(C)$;
6 $\quad r \leftarrow C_O[0]$;
7 $\quad C_O \leftarrow C_O \setminus \{r\}$;
8 $\quad \phi_{IB}^d \leftarrow \phi_{IB}^d \cup descendExplanationTree(r, C_O, D, \emptyset)$;
9 $\phi_{QB}^d \leftarrow getQueryBasedPart(d)$;
10 **return** $\phi_H^d$;

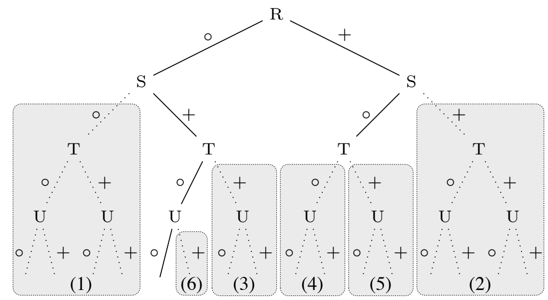

**Figure 3: Determining c-tuple labels**

The remainder of the discussion focuses on how our algorithm proceeds, based on the above example.

First, in deciding whether to assign label $\circ$ or $+$ to the c-tuple in $R$, we first observe that we have $\langle \mathsf{a'}, \mathsf{b}\rangle \in R$. At that point of the algorithm, we however continue to consider both options, because we cannot verify that any label assignment for the remaining relations (including the worst-case assignment) in combination with $\circ R$ will yield a lower global cost than using $+R$ combined with any possible label assignments for the remaining relations (including the best-case assignment). More formally, we cannot prune the right subtree of $R$ (the one assigning $+$ as label for $R$) because we cannot verify that

$$\begin{array}{l} minCost(R) + maxCost(S) + maxCost(T) + maxCost(U) \\ \leq maxCost(R) + minCost(S) + minCost(T) + minCost(U) \end{array} \quad (2)$$

As a consequence, we further process both subtrees. Let us denote the left and right subtrees of $R$ as $LT_R$ and $RT_R$, respectively.

In $LT_R$, we do not have any tuple in $S$ that would join with $\langle \mathsf{a'}, \mathsf{b}\rangle \in R$ while having S.C = c', hence, the left subtree of $LT_R$ can be pruned due to the lack of necessary source data (Step (1)).

In $RT_R$, it is true that $\langle \mathsf{b'}, \mathsf{c'}\rangle \in S$ (and the tuple inserted to $R$ can be made such that it joins with this tuple), so both $\circ$ and $+$ need to be considered. To verify if we can prune the right side, we verify if

$$\begin{array}{l} minCost(S) + maxCost(T) + maxCost(U) \\ \leq maxCost(S) + minCost(T) + minCost(U) \end{array}$$

Let us assume this holds. As a consequence, we prune the right subtree of $RT_R$, denoted $RT_{RT_R}$ (Step (2)).

We now move to the level of $T$, where we further investigate the subtrees $RT_{LT_R}$ and $LT_{RT_R}$. For $RT_{LT_R}$, we find $\langle \mathsf{a}, \mathsf{b}, \mathsf{c'}, \mathsf{d'}\rangle \in T$ so we have two candidate branches, of which we can prune the right one if $minCost(T) + maxCost(U) \leq maxCost(T) + minCost(U)$. This holds according to Equation 1, hence, we prune the right subtree in Step (3).

By this last pruning, the worst case we have considered so far (Equation 2) is excluded and the worst case becomes $minCost(R) + maxCost(S) + minCost(T) + maxCost(U)$. We compare this new worst case to the unchanged best case, which does not allow us further pruning.

In processing $LT_{RT_R}$, we check if $T$ contains a tuple that joins with $+R(a', \mathsf{b'}), \circ S(\mathsf{b'}, \mathsf{c'})$. No such tuple exists in $T$, so the left subtree can be pruned (Step (4)). As a consequence, the best case of Equation 2 updates to $maxCost(R) + minCost(S) + maxCost(T) + minCost(U)$ We now verify hat

$$\begin{array}{l} minCost(R) + maxCost(S) + minCost(T) + maxCost(U) \\ \leq maxCost(R) + minCost(S) + maxCost(T) + minCost(U) \end{array}$$

so we prune the remainder of $RT_{LT_R}$ (Step (5)).

| 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 | $\Pi_{p.name}(\sigma_{c.sector>97}(c) \bowtie_{c.sector=w.sector} w \bowtie_{w.name=sp.witness} sp \bowtie_{sp.hair=p.hair \wedge sp.clothes=p.clothes} p)$ |
| MOV1 | $\Pi_{l.title}((\sigma_{l.year<2009}(l) \bowtie_{l.title=r.title} \sigma_{r.rating>9}(r)) \backslash (\Pi_{t.title}(\sigma_{t.rank<10}(t))))$ |
| MOV2 | $\Pi_{loc.loc1}(\sigma_{l.year>1994}(l) \bowtie_{l.mid=rel.mid} rel \bowtie_{rel.lid=loc.lid} loc \bowtie_{l.title=r.title} \sigma_{r.rating\geq 9}(r))$ |
| MOV 3 | $\alpha_{loc.loc1,AVG(r.rating)}(\sigma_{l.year>1994}(l) \bowtie_{l.mid=rel.mid} rel \bowtie_{rel.lid=loc.lid} loc \bowtie_{l.title=r.title} \sigma_{r.rating\geq 9}(r))$ |
| GOV1 | $\Pi_{e.agency,e.bureau,e.title,e.desc,e.totalamount,es.substage}(e \bowtie_{e.eid=es.eid} es \bowtie_{es.sid=s.sid} (\sigma_{s.ln=`Pelosi'}(s) \cup \sigma_{s.ln=X}(s))$ |
| GOV2 | $\alpha_{s.competed,avg(s.obligated)}(\sigma_{s.obligated>10000}(\sigma_{s.competed=`yes'}(s) \cup \sigma_{s.competed=`no'}(s)))$ |
| 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**

Our final verification (Step (7)) identifies that $\langle \mathsf{a,u} \rangle \in U$ satisfies all necessary conditions and it trivially holds that $minCost(U) \leq maxCost(U)$ so our final c-tuple label assignment is $\circ R, +S, \circ T, \circ U$.

In the sample algorithm run discussed above, we have seen that it is possible to prune a potentially large fraction of the search space. In the worst case, we would have to make $\sum_{i=0}^{n-1} 2^i$ cost comparisons, e.g, 15 in the example. Instead, we only performed 6 such comparisons to obtain the optimal solution.

In our current implementation, the above seach-space reduction algorithm is the only efficiency optimization when determining explanations. There is, however, potential to further improve the overall efficiency of our algorithm that we will explore in the future.

**Cost model.** We very briefly describe the cost model we use in our implementation. The goal of the work we present here is not to define "the best" cost model and we leave it to future work to investigate further reasonable cost models. In general, we postulate that the cost $maxCost(X)$ for a relation $\mathsf{X}$ should be higher the more difficult it becomes in practice to modify the source database $D$ such that the constraints of the c-tuple on $\mathsf{X}$ are satisfied. Opposed to that, $minCost(X)$ should be lower the more trusted the reused data in $D$ that satisfies the c-tuple's constraints is.

In our implementation, we consider all source tables are equally trusted, hence $minCost(X) = 0$ for any relation $X \in D$. In determining $maxCost(X)$, we take into account both key and foreign key constraints pertinent to $X$ as follows:

DEFINITION 13 ($maxCost(X)$). *The maximum cost of a relation $\mathsf{X}$ computes as $maxCost(X) = 1 + |K| + |FK| + \#cond$, where $K$ is the set of key attributes of $\mathsf{X}$ on which the corresponding c-tuple $t_C$ applies a constraint, $FK$ is the set of foreign-keys of $\mathsf{X}$, and $\#cond$ is the number of conditions defined in $t_C.cond$.*

EXAMPLE 17. *For the constraints of Example 9 and the cluster of Example 15, $maxCost(R) = 5$, $maxCost(S) = maxCost(T) = 4$, $maxCost(U) = 3$, thus satisfying Equation 1.*

## 5. EVALUATION

We implemented *Conseil*, Artemis [13], and Why-Not [4] in Java 1.6.. We chose these algorithms for comparison as they represent the state-of-the-art for producing instance-based explanations and query-based explanations, respectively. 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 use in our implementation. For Artemis, we additionally use Minion (http://minion.sourceforge.net), as done in the original implementation of Artemis.

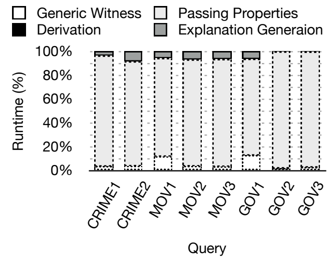


**Figure 4: Phase-wise runtime**

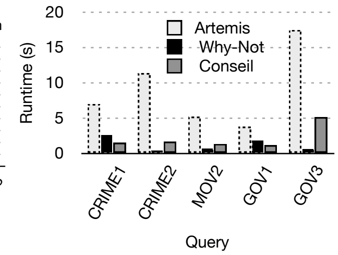


**Figure 5: Alg. comparison**

The results reported here are based on eight queries, summarized in Table 3. They are issued from three different scenarios, identified based on the query name prefix. 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.

**Experiment 1: Phase-wise runtime of *Conseil*.** Our first experiment studies how much time *Conseil* spends on each of its four phases. To this end, we randomly select ten missing-answers for each query, and run *Conseil* once for each query. Figure 4 reports the average percentage of overall runtime spent on each phase for all queries. Obviously, determining passing properties is the most time-consuming task. This is due to the fact that this phase traces actual data from $D$ through all operators of $Q$, which requires the processing of several SQL queries in our implementation (one for each subquery of $Q$ rooted at a potentially blocking operator). The complexity of witness generation depends on the size of $Q$, which we also observe in our experimental evaluation (the "small" query CRIME1 take 49 ms whereas the "large" query GOV2 takes 176 ms). Derivation time is negligible in all cases. Concerning hybrid explanation generation, we observe significant differences in terms of runtime, e.g., CRIME1 requires 55 ms whereas CRIME2 takes 162 ms. These differences are due to varying cluster sizes and subsequent c-tuple label-assignment efficiency.

**Experiment 2: Runtime comparison.** Next, we compare the runtime of *Conseil* with the runtime of Artemis and Why-Not. We limit the evaluation to those queries supported by all three algorithms, i.e., CRIME1, CRIME2, MOV2, GOV1, and GOV3. When first running Artemis, we observed that it takes prohibitively long for it to compute all instance-based explanations. The reason for this is that Artemis will essentially form the cross-product of all joined relations, in which each tuple is then further processed. For instance, in CRIME1, 4,764,484 tuples need processing. To obtain results in reasonable time, we thus decided to add constraints to the debugging scenarios of Artemis, trusting all but one table in all scenarios where necessary (i.e., all but MOV2 and MOV3). Figure 5 shows the runtime results. When trust was needed for Artemis, we report the best debugging scenario here.

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, as mentioned above. 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,

| Query | Missing-answer | Artemis | Why-Not | *Conseil* |
|---|---|---|---|---|
| CRIME1 | ⟨Roger, Laugh⟩ | 0 | 1 | 2 |
| CRIME2 | ⟨Conedera⟩ | 424 | 1 | 2 |
| MOV2 | ⟨Germany⟩ | 27 | 1 | 2 |
| GOV1 | ⟨Edu, ?, ?, ?, $v_4$, Enacted, $v_4 \neq NULL$⟩ | 2 | 1 | 2 |
| GOV3 | ⟨Pelosi, Nancy, ?, CA⟩ | 854 | 0 | 2 |

**Table 4: Explanations returned by different algorithms**

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 in this paper, we report in Table 4 the number of explanations each algorithm returns on the same set of queries as Experiment 2, but for a specific missing-answer. 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. Note that in general, *Conseil* can return more than two explanations, which did however not occur in the use cases described in this paper.

# 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, making it applicable to a wide range of practical queries. 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.

In the future, we plan to also consider side-effects and more general debugging scenarios (more than one query, more than one missing-answer). We also plan to further study efficiency improvements and cost models and to make a more thorough usability study.

**Acknowledgements.** 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. A. Hernández, G. Koutrika, R. Krishnamurthy, L. Popa, and R. Wisnesky. Hil: a high-level scripting language for entity integration. In *International Conference on Extending Database Technology (EDBT)*, pages 549–560, 2013.

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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.

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

[19] 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.