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.

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

Assume the tuple \( \langle P1, \text{Car}, v_1 \rangle \) 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 \( \text{MAX}(R.\text{Rating}) \leq 2 \) 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 \( \text{MAX}(R.\text{Rating}) < 5 \).

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 \( \langle P3, \text{Bus}, 0 \rangle \). Indeed, an instance-based explanation would have to insert tuple \( \langle P3, \text{Bus}, \text{US} \rangle \) 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-monotonic 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 \(t = (a_1, \ldots, a_n, \text{cond})\) 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 \(t = (a_1, \ldots, a_n)\) belongs to, we use \(R(a_1, \ldots, a_n, \text{cond})\), 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 \((E, Q, Q(D), D, C)\), where \(Q\) is a set of queries, \(Q(D)\) 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 \(Q(D)\), and \(C\) 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 \(t.1.\text{cond} \land t.2.\text{cond}\).

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 \(t_1 = (\{P1, \text{Car}, v_1, v_1 \neq \text{‘UK’}\}, D)\), \(t_2 = (\{P1, v_2, \text{US}, v_3\text{ LIKE ‘C%’}\}, D)\), and \(t_3 = (\{P1, \text{Car}, \text{US}\}, D)\). 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 \(t.1.\text{cond} \land t.3.\text{cond}\). 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 \((\{P1, \text{Car}, \text{US}\}, D)\) for which it is easy to verify that both \(t_1.\text{cond}\) and \(t.2.\text{cond}\) hold. Hence, \(t_1\) is compatible with \(t_2\) (and vice versa).

Definition 4 (Labeled c-Tuple). A labeled c-tuple \(t = (L(a_1, \ldots, a_n, \text{cond}), 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 \(Q(D)\) while satisfying constraints \(C\). The syntax of \(\phi_{IB}\) is:
\[ \phi_{IB} = \{\{T_1, \ldots, T_n\}\}_{A} \]

\[ C := L(a_1, \ldots, a_n, \text{cond}) \]

\[ \text{group} := \text{group}(\text{acopw}) \]

\[ A := \text{group}(\text{agg}\text{group} \land \text{agg}\emptyset) \]

\[ \text{agg} := \text{agg}\text{agg}(A)_{a\text{Con}} \]

where \( L(a_1, \ldots, a_n, \text{cond}) \) refers to Def. 4, \( a \) is an attribute, \( v \) a value (constant or variable), \( \text{cop} \in \{=, >, \leq, \geq\} \), \( \text{agg}\text{F} \) is an aggregation function over attribute \( a \), and \( a\text{Con} \) a condition on the aggregated value of \( a \). The semantics of \( \phi_{IB} \) describe the sequence of operations \( \{T_1, \ldots, T_n\} \) 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 \( T \) either corresponds to a labeled c-tuple \( C \) or again \( \phi_{IB} \), necessary for nested queries.

**Example 5.** The instance-based explanation of Ex. 1 is defined as follows, assuming that all ratings for \( P \) are above 2.

\[ \phi_{IB} = \left\{ \left[\{\text{Product}(P1, \text{Car}, \text{US}), \text{Ratings}(P1, v_1, v_1 > 2), +\text{Ratings}(P1, v_2, v_2 <= 2)\}\} \right] \text{PrfID} = P1, \text{Name} = \text{Car} \right\} \]

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 \( \text{op}_i \in \phi_{QB} \) is responsible for pruning missing-answers of \( E \) from \( Q(D) \) and satisfies \( C \). An operator \( \text{op}_i \) is responsible for pruning a missing answer \( e \in E \) if data relevant to produce \( e \) is in the input of \( \text{op}_i \), but not in its output.

The definition leaves open the choice of one or more operators in \( \phi_{QB} \), 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 \( e = (P1, \text{Car}, v_1) \), data relevant to produce \( e \) includes the tuple \( (P1, \text{Car}, \text{US}) \in \text{Products} \) and all tuples in \( \text{Ratings} \). These data find “successors” in the output of all operators of the query, except for the selection \( \sigma_{\text{Max}(\text{R}\text{.Rating}) \leq 2} \). Consequently, \( \phi_{QB} = \{\{\text{Max}(\text{R}\text{.Rating}) \leq 2\}\} \).

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 \( Q \) into a set of queries \( Q^{'} \) such that all missing tuples in \( E \) occur in \( Q^{'}(D) \) for a given source instance \( D \) and \( C \) is satisfied.

**Example 7.** A modification-based explanation for our example replaces the existing \textit{HAVING}-clause with \textit{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 \( \phi_{H} \) for a debugging scenario \( S \) is a 3-tuple \( \phi_{H} = \phi_{IB}, \phi_{QB}, \phi_{MB} \) s.t. the conjunction of all \( \phi_i \) 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, \( \phi_{QB} \) or \( \phi_{MB} \) 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

\[ \phi_{IB} = \left\{ \left[\{\text{Product}(P3, \text{Bus}, \text{CH}), +\text{Ratings}(P3, 0)\}\} \right] \text{PrfID} = P3, \text{Name} = \text{Bus} \right\} \]

**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 \( \sigma \), projection \( \Pi \), union \( \cup \), and set difference \( \setminus \)) with the restriction that it only supports one set difference. It also supports aggregation \( \alpha \). 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 \( C \).

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

Conseil operates in four main phases: First, (1) it computes a generic witness \( \Phi \) 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 \( S_{\text{Conseil}} \) with \( e = (a, c, d) \), \( Q \) and \( D \) as in Fig. 2, \( Q(D) = \{a, c, d\} \), 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 \( \Phi_I \) and a query-based component \( \Phi_Q \). 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 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., \( \sigma, \Pi, \cup \), and \( \setminus \).

To define the generic witness \( \Phi \), we first need to distinguish between missing-tuple constraints, subsequently called \textit{mt-constraints}, and query-constraints, or \textit{q-constraints} for short.

**Definition 9.** (Mt-constraint). An \textit{mt-constraint} is a constraint that, given \( e \) and \( Q \), can be identified as being imposed on the lineage [6] \( D\ast \) of \( e \) w.r.t. \( Q \) by the missing-tuple \( e \).

**Definition 10.** (Q-constraint). A \textit{q-constraint} is a constraint that, given \( e \) and \( Q \), can be identified as being imposed on the lineage \( D\ast \) of \( e \) w.r.t. \( Q \) by the query \( Q \).

**Example 10.** For Ex. 9, one \textit{mt-constraint} is \( R.A = a' \) whereas \( U.B = u \) illustrates a \textit{q-constraint}.
each explanation conforms to. Similarly to a hybrid explanation, a generic witness

\[
\Phi \equiv \left( \Phi_i \setminus \{ R_j \mid \text{cond}_{j} \land (a \ cop) \} \right) \cup \left( R_k \setminus \{ a \mid \text{cond}_{k} \land (a \ cop) \} \right)
\]

where \( R_j \) is a table reference, \( a \) is an attribute of table \( R_j \), \( \text{cop} \) is a comparison operator, and \( e \) is an constant.

\[
\Phi_q = \{ \Phi_i \setminus \{ R_j \mid \text{cond}_{j} \} \}
\]

Table 1: Derivation rules
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 \( \Phi_1 \) and each operator in \( \Phi_2 \) is assigned an annotation. The set of possible annotations is the set \{\( \Box \), \( \Lambda \), \( \circ \)\}, 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 \textit{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 \textit{annotate} and return the final annotated canonical tree \( T_A \).

In discussing \textit{annotate}, we invite the reader to follow the discussion on Fig. 2. We assume a function \textit{getCTuple} \((R, \Phi)\) that extracts the \( c \)-tuple in the generic witness \( \Phi \) 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.

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

The only operators that introduce \( q \)-constraints are \( \sigma \) and \( \land \). 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. We then comment on how to include union and aggregation.

DEFINITION 11 (GENERIC WITNESS \( \Phi \) FOR SPJD-QUERIES).

A generic witness \( \Phi = (\Phi_1, \Phi_2) \) for a SPJD-query is a 2-tuple of sets \( \Phi_1 \) and \( \Phi_2 \), where \( \Phi_1 = \{ ti \mid \text{relation}(ti) \in D^* \text{ w.r.t. } e \text{ and } Q \land \text{cond} \} \) and \( \Phi_2 = \{ \text{op}_i \mid \text{op}_i \in Q \land \text{op}_i \in \{ \sigma, \land, \} \} \).

EXAMPLE 11. The generic witness for the debugging scenario of Ex. 9 has \( \Phi_1 = \{ R(a, b), S(\tau, c), T(\tau, \tau, d), U(\tau, \tau) \} \) and \( \Phi_2 = \{ \text{getCTuple}(R, \Phi), a, b, c \} \). In case a query involves union operators, we create one generic witness for each alternative. For instance, \( Q = (R \land S) \cup T \) 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]). Given a missing tuple \( (b, 3, c) \) and a query \( Q = \Pi_{B, A, C}(\text{COUNT}(A) \text{ASA}(R \land S)) \), we obtain \( \Phi_1 = \{ \{ R(v_{ba}, b) \mid _{B, \text{COUNT}(A) = 3}, S(b, c) \} \} \)

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
Given a generic witness \( \Phi = \{ \{ t_1, \ldots, t_n \}, \{ \alpha_1, \ldots, \alpha_m \} \} \)
for a SPJ-\( \star \) query, where \( t_i = R(a_1, \ldots, a_{|\mathcal{A}|}, \text{cond}) \) and \( \alpha \in \{ \sigma, \geq, \leq \} \), we determine a derivation \( \Phi' = \{ \Phi'_{\text{q}}, \Phi'_{\text{op}} \} \) by applying the derivation rules summarized in Tab. 1. In general, a derivation is an iterative process that transfers one \( q \)-constraint of \( \alpha \in \Phi_{\text{Q}} \) into an \( m \)-constraint.

We first apply the derivation rules to translate all passing operators of \( T_A \) into \( m \)-constraints and corresponding \( c \)-tuples to be added to \( \Phi_{\text{Q}} \). 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 Consee, 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., \( \sigma_{\text{MAX}}(R, \text{Ranking} \leq 2) \)) either into the \( c \)-tuples belonging to the grouped and aggregated sub-query (for \( \text{MIN} \) and \( \text{MAX} \) ) or to \( \text{agg} \) itself (for \( \text{COUNT}, \text{SUM}, \text{AVG} \)). 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), \( \triangleright_{R.B=S.B} \) (line 2), \( \triangleright_{R.B=T.B} \) (line 3), and finally both joins (line 4).

### 4.4 Step 4: Explanation Computation

In its final step, Consee 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 pre-
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.

**Acknowledgements.** Fundamental ideas of the Conseil algorithm have been developed in collaboration with Tim Bellhomme. I also thank Hannes 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**


