Explaining Missing Answers to SPJUA Queries

Melanie Herschel
Universität Tübingen
72076 Tübingen, Germany
melanie.herschel@uni-tuebingen.de

Mauricio A. Hernández
IBM Research – Almaden
San Jose, CA, 95120, USA
mauricio@almaden.ibm.com

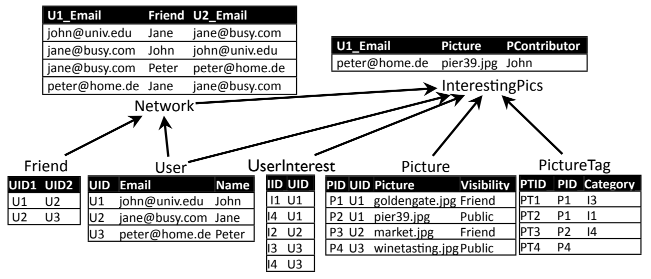
ABSTRACT

This paper addresses the problem of explaining *missing answers* in queries that include selection, projection, join, union, aggregation and grouping (SPJUA). Explaining missing answers of queries is useful in various scenarios, including query understanding and debugging. We present a general framework for the generation of these explanations based on source data. We describe the algorithms used to generate a correct, finite, and, when possible, minimal set of explanations. These algorithms are part of Artemis, a system that assists query developers in analyzing queries by, for instance, allowing them to ask why certain tuples are not in the query results. Experimental results demonstrate that Artemis generates explanations of missing tuples at a pace that allows developers to effectively use them for query analysis.

1. Introduction

A common scenario faced by SQL programmers involves asking why one or more tuples are missing from the results of a query. One might wonder why, for instance, the result of a query is empty or why a query did not return certain tuples. In the case when queries are used to define multiple views, one may ask why, for instance, an employee information is missing from both the employee register and the payroll views. Often, the first reaction of programmers is to review the query itself since the explanation could be that a filter is too restrictive, or an inner-join should be an outer-join. However, if the expressions in the queries appear to be correct, then the next step is exploring the data sources to figure if data that maybe combined to yield the desired tuples are indeed there. We call this latter kind of explanations, in which data is used to explain missing answers, *instance-based explanations*.

We present a *missing answers explanation framework* implemented in Artemis [11], a system for debugging and validating SQL queries using data. Artemis' algorithms are able to generate explanations for a *set* of missing tuples over a *set* of queries that include selection, projection, join, union, aggregation, and grouping (SPJUA queries for short). Missing values are entered by users as a "pattern" representing the missing tuples. These tuple patterns can contain the actual expected values for certain attributes, and "labeled nulls" representing unknown values.



Network

U1_Email	Friend	U2_Email
john@univ.edu	Jane	jane@busy.com
jane@busy.com	John	john@univ.edu
jane@busy.com	Peter	peter@home.de
peter@home.de	Jane	jane@busy.com

InterestingPics

U1_Email	Picture	PContributor
peter@home.de	pier39.jpg	John

Friend

UID1	UID2
U1	U2
U2	U3

User

UID	Email	Name
U1	john@univ.edu	John
U2	jane@busy.com	Jane
U3	peter@home.de	Peter

UserInterest

IID	UID
I1	U1
I4	U1
I2	U2
I3	U3
I4	U3

Picture

PID	UID	Picture	Visibility
P1	U1	goldengate.jpg	Friend
P2	U1	pier39.jpg	Public
P3	U2	market.jpg	Friend
P4	U3	winetasting.jpg	Public

PictureTag

PTID	PID	Category
PT1	P1	I3
PT2	P1	I1
PT3	P2	I4
PT4	P4	

Figure 1: The PhotoShare database (sample)

Consider for example a simple debugging scenario for a social network application. Fig. 1 shows an excerpt of a database used by an application called PhotoShare that models a network of friends sharing pictures. The five relations at the bottom are the source relations and the two at the top are relational views. Arrows indicate which relations appear in the FROM clause of the view definitions. The source schema stores information about PhotoShare users including their friends, pictures taken, and tags about their interests. Each picture can also be tagged as relevant to interests.

The *Network* view associates the e-mail addresses of people connected by the *Friend* table. This association is bi-directional and is computed by the union of two almost identical SPJ queries (one for each direction). *InterestingPics* finds, for each user, a list of shared pictures whose tag matches one of the user's interests. For example, peter@home.de appears in this view because the picture pier39.jpg, contributed by John, was tagged as I1, an interest that Peter shares. The *InterestingPics* query is a union between shared pictures visible to everybody and pictures that are only visible to friends. Fig. 7 (Appendix) shows the actual SQL view definitions.

As an initial example, assume the query programmer wants to know why Peter does not appear as a friend of John in *Network*. The user queries Artemis by adding the following tuple pattern into *Network*. The tuple pattern contains constant values and labeled nulls, denoted by a $ sign followed by the name of the null.

$$t_1 = Network\,\langle\text{'john@univ.edu'}, \text{'Peter'}, \text{\$friend-email}\rangle$$

Fig. 2 shows four explanations for t_1 (each row represents one explanation). Each explanation has as many attributes as there are in all source relations and can be thought of as the result of joining tuples from each source relation. In our example, the query is a join between *Friend* and two instances of *User* and, thus, each explanation contains a *Friend* tuple and two *User* tuples. Notice that some of these tuples already exist in the source relations (illustrated with the lighter background) while others do not exists and represent the "missing" source tuples that, if added, would produce t_1. Consider, for instance, the last explanation in Fig. 2 (row 4). To

	Friend F		User U1			User U2		
	F.UID1	F.UID2	U1.UID	U1.Email*	U1.Name	U2.UID	U2.Email*	U2.Name*
1	U1	=U2.UID	U1	john@univ.edu	John	=F.UID2	$friend-email	Peter
2	U1	U3	U1	john@univ.edu	John	U3	peter@home.de	Peter
3	=U1.UID	U1	=F.UID1	$friend-email	Peter	U1	john@univ.edu	John
4	U3	U1	U3	peter@home.de	Peter	U1	john@univ.edu	John

Figure 2: Explanations creating t_1

create t_1, a tuple $\langle \mathsf{U3}, \mathsf{U1} \rangle$ must be inserted into *Friend*. That new tuple will join with the two existing tuples in *User* shown in row 4. Another possibility, described in row 1, is to insert a new tuple into *Friend* and a new tuple into *User*. Those tuples include a condition, namely $F.UID2 = U2.UID$, needed to join the tuples to produce t_1.

In a little more complex example, our query programmer needs to know why john@univ.edu does **not** appear in *InterestingPics*. Notice that we can extract several possible high-level explanations by studying how source data is joined. Maybe John is no longer a PhotoShare user, or does not have a declared interest. Perhaps none of the users is sharing a photo tagged with John's interest. Or, even if they have such a photo, the photo is not visible to John.

To explain this missing value, the user enters a new tuple pattern into *InterestingPics*:

$$t_2 = \mathit{InterestingPics}\langle \text{'john@univ.edu'}, \$\text{picture}, \$\text{friend-name} \rangle$$

Further, assume the user wants to ensure john@univ.edu is a friend of the person contributing the picture. This is easily done by entering a new tuple pattern

$$t_3 = \mathit{Network}\langle \text{'john@univ.edu'}, \$\text{friend-name}, \$\text{friend-email} \rangle$$

Notice that $friend-name is present in both t_2 and t_3, meaning that both tuples share the same (unknown) value. In a sense, the specification of tuple patterns resembles Query-By-Example [16].

To explain t_2 and t_3 simultaneously, we compute the cross product of all explanations for t_2 and t_3. In our example, Artemis produces 7 explanations for t_2 and 67 explanations for t_3, resulting in 469 explanations in the cross product.

Artemis provides several ways to reduce the number of explanations. Users can, for example, request that no new tuples be added to the *User* table. We can also restrict or minimize the "side-effect" an explanation creates. For example, consider what happens if we use the explanation for t_1 in row 2 (Fig. 2) and insert a tuple *Friend*$\langle \mathsf{U1}, \mathsf{U3} \rangle$. If that tuple is inserted, t_1 will be generated as requested. However, *InterestingPics* $\langle$peter@home.de, goldegate.jpg, John$\rangle$ will also be generated as a "side-effect". To avoid these cases, users can request Artemis to only produce explanations that minimize, or even avoid changes on a view. Minimizing side-effects in our example reduces the number of explanations for t_2 to 4 and the number of explanations for t_3 to 13, for a new total of 52 explanations.

Our work is largely inspired by the work in [12] but it extends that work in non-trivial ways. Our contributions specifically are:

Framework. We present a framework for instance-based explanation generation and study properties of the generated output such as universality and minimality.

Explaining a set of missing tuples over a set of SPJUA queries. Artemis provides explanations over a set of SPJUA queries ([12] only covers a single SPJ query) for a set of missing tuples ([12] provides explanations for one tuple at a time). One could argue that it is possible to trivially extend the algorithm in [12] to support multiple SPJU queries and explain multiple tuples. However, we notice that for the example in this section, [12] produces 25 explanations for t_2 alone and 396 for t_3 alone. Such a trivial extension would produce 9,900 explanations for this simple scenario.

Side-effects are considered by Artemis (and not by [12]).

Correctness. We compute explanations by encoding the problem into a set of constraints which are passed to a constraint solver. An explanation is returned to the user only if the constraint solver can find a satisfiable solution for its constraints. (as we discuss in Appendix C, [12] can return explanations that do not satisfy constraints when explanations require nulls, inequalities, or unique constraints).

We discuss some related work next. We define a framework for explanation generation for SPJU queries in Sec. 3. Sec. 4 describes the algorithm for producing explanations for SPJU queries and Sec. 5 extends this algorithm to grouping and aggregation. In Sec. 6 we evaluate our algorithms, before we conclude in Sec. 7.

2. Related Work

The instance-based explanations extend the traditional data provenance research, where the problem is to determine what data and/or transformations led to existing tuples (see a survey in [6]). The idea of extending the computation of provenance information of missing tuples has recently been explored by [12] and [5].

The Missing-Answers algorithm [12] computes instance-based explanations given a single missing tuple and a single SPJ query. We highlighted our extensions to these capabilities in Sec. 1.

Explanations for missing tuples can also be created by analyzing the query operations [5]. Given a missing tuple t, [5] first identifies tuples in the source database D that contain the constant values in t and are not part of the lineage 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 our work, we do not consider query-based explanations, but note that they complement instance-based explanations.

A framework that models both types of explanations based on functional causality has recently been proposed [15]. Opposed to that, our framework mainly focuses on a concise definition of input and output for the generation of instance-based explanations.

Our algorithm relies on conditional tuples, or c-tuples for short. C-tuples and the semantics and evaluation of relational queries over tables containing them were defined in [13] for unions of conjunctive queries. Aggregation and grouping using c-tuples is discussed in [14]. The queries we use to generate explanations based on c-tuples (Sec. 4) make use of techniques developed in previous work on view update and maintenance over c-tuples [17, 18].

A related research area is the generation of test databases [3]. Given a query and a relational instance generated by that query, the goal is to create a source database that produces the generated instance. Indeed, generating source data for a given result of a query is similar to the first step of our algorithm that determines what data should be present in the sources to produce the missing tuples, and the use of a constraint solver to produce correct test data was also explored in [3]. The generation of explanations is different to the generation of test databases as an explanation combines existing data with new data, considers side-effects, and we consider multiple views and additional constraints as well.

3. SPJU Explanation Framework

The basic Artemis explanation framework takes as input a *debugging scenario* specification and returns a number of instance-based explanations. We define our input as follows:

Definition 1 (Debugging Scenario). *A* debugging scenario *$\mathcal{S}$ is defined by a 5-tuple $\langle \mathcal{Q}, \mathcal{Q}_{im}, \mathcal{Q}_m, D, E \rangle$. $\mathcal{Q}$ is the set of queries that we are debugging. D represents the source database for $\mathcal{Q}$ and we use $\mathcal{Q}(D)$ to denote the set of tuples produced by $\mathcal{Q}$. E represents a set of* c-tuples *that encode missing tuples that do not exists in $\mathcal{Q}(D)$ and that require an explanation. We further model a set $\mathcal{Q}_v$ of identity views that each mirror a source relation, i.e., for each relation $R_i \in D$ a view $V_i(X)$:- $R_i(X)$ is added to $\mathcal{Q}_v$.*

$\mathcal{Q}_{im} \subset \mathcal{Q} \cup \mathcal{Q}_v$ is the set of "immutable" queries on which no side-effects are admissible as part of the explanations. $\mathcal{Q}_m \subset \mathcal{Q} \cup \mathcal{Q}_v$ is the set of queries in which only the minimal number of side-effects is accepted for an explanation. Further, $\mathcal{Q}_m \cap \mathcal{Q}_{im} = \emptyset$.

In the *PhotoShare* debugging scenario of Sec. 1, $\mathcal{Q} = \{Q_{\text{Network}}, Q_{\text{InterestingPics}}\}$ (the queries defining the views), $D = \{$ *Friend*, *User*, *UserInterest*, *Picture*, *PictureTag*, *Network*$\}$, and $E = \{t_2, t_3\}$. *Network* is in D because it is a source in *InterestingPics*. We also set $\mathcal{Q}_m = \{InterestingPics\}$ and $\mathcal{Q}_{im} = \{V_{\text{User}}\}$. Here $V_{\text{User}} \in \mathcal{Q}_v$ represents the identity view over the source relation *User*. These identity views allow us to treat all source relations as queries to which we can restrict side-effects.

Note that $\mathcal{Q}_{im}$ is similar in spirit to the concept of table trust in [12]. However, our framework is a bit more flexible and allows marking any subset of the data (both in the source relations or in the query results) as immutable, based on the data being identified by a query. On the other hand, [12] allows marking attributes in relations as immutable, avoiding the generation of explanations that otherwise update these attributes. Our framework only explains missing tuples by inserting rows and, thus, does not support constraints to avoid updates.

Previously, we informally described t_2 and t_3 as tuple patterns. We now formally define them as c-tuples:

DEFINITION 2 (C-TUPLE). *A c-tuple is a tuple $\langle a_1, a_2, \ldots, a_n, cond\rangle$ such that every value a_i is either a constant or a labeled null. The attribute cond is a boolean expression.*

C-tuples populate conditional tables (c-tables) and a single c-tuple in the extension of a c-table stands for any tuple whose valuation satisfies *cond*.

Using c-tuples on E allows users to put constraints on the missing tuples that need explanation. For example, assume we have a view Employees and we want to know why an employee named 'John' is not in the result. We might not remember how to spell John's last name but we might know that John is not older than 25. We can specify this as follows:

$$t = Employee\langle \text{'John'}, \$\text{lastname}, \$\text{age}, \ldots, \$\text{age} < 25\rangle$$

Since our debugging scenarios involve multiple queries, we add the name of the query (or relation) to each c-tuple to clarify the context. As discussed next, c-tuples are also used to explain the missing tuples. These explanations involve c-tuples that already exist, combined with new c-tuples. To distinguish between c-tuples that exist and c-tuples that do not exist, we add a '+' sign in front of c-tuples that do not exist. The output of our system is a set of explanations, called explanation set.

DEFINITION 3 (EXPLANATION). *Given a debugging scenario $\mathcal{S}$, an explanation ψ is a set of labeled c-tuples. The label of a c-tuple describes whether the c-tuple represents an existing or a non-existing tuple. ψ describes how data existing in D combines with c-tuples in order to generate* **all** *missing tuples in E while satisfying the constraints on side-effects specified by $\mathcal{Q}_{im}$ and $\mathcal{Q}_m$.*

DEFINITION 4 (EXPLANATION SET). *Given a debugging scenario $\mathcal{S}$, an explanation set $\Psi = \{\psi_1, \psi_2, \ldots \psi_n\}$ is a finite set of explanations that represents all possible explanations w.r.t. $\mathcal{S}$.*

EXAMPLE 1. *Consider the following relations $R(A,B)$ and $S(A,B,C)$, and the conjunctive query Q: $Q(A,B)$:- $R(A,B), B > 5$; $Q(A,B)$:- $S(A,B,C), C < 5$; $Q(A_1,B_2)$:- $R(A_1,B_1), S(A_1,B_2,C)$.*

R	A	B
	a	1
	b	6
	d	3

S	A	B	C
	c	3	4
	a	9	7

Q(D)	A	B
	b	6
	c	3
	a	9

Assume we want to explain why $\langle d, 6\rangle$ is not in the result of Q. The explanation set is (with N_i denoting labeled nulls):

ψ_1 :	$+R\langle d,6,true\rangle$	satisfies $R(A,B), B > 5$
ψ_2 :	$+S\langle d,6,N_1,N_1 < 5\rangle$	satisfies $S(A,B,C), C < 5$
ψ_3 :	$R\langle d,3\rangle, +S\langle d,6,N_1,true\rangle$	satisfies $R(A_1,B_1), S(A_1,B_2,C)$
ψ_4 :	$+R\langle d,N_2,true\rangle, +S\langle d,6,N_3,true\rangle$	satisfies $R(A_1,B_1), S(A_1,B_2,C)$

Having defined our framework, let us now focus on an algorithm that implements the framework.

4. SPJU Explanation Algorithm

Before we delve into the details of our algorithm, we provide further definitions. These definitions are used to guarantee that the output explanation set is finite and covers all possible explanations.

4.1 Definitions

In general, the number of explanations for a given scenario is infinite. One could, for example, propose the following as an additional explanation for Ex. 1:

$$\psi_5 : +R\langle d,6,true\rangle, +S\langle x,4,5,true\rangle$$

This explanation is, however, already covered by ψ_1 and contains an unnecessary $+S$ c-tuple. To avoid such redundant explanations, our algorithm only generates explanations that match an *explanation pattern*.

DEFINITION 5 (EXPLANATION PATTERN). *An explanation pattern P is a finite set of c-tuples that is sufficient to yield all tuples in E, i.e., $\mathcal{Q}(P) \supseteq E$.*

To determine if an explanation matches an explanation pattern, we use the following definition. Remember that both explanations and patterns are sets of c-tuples.

DEFINITION 6 (MATCHING C-TUPLE SETS). *Two c-tuples t_1 and t_2 match if (i) they are instances of the same c-table, (ii) we can unify the corresponding values in t_1 with the values in t_2, and (iii) $t_1.cond \wedge t_2.cond$ is satisfiable. Two sets of c-tuples "match" if (i) for each c-tuple in the first set we can find a match in the second set, and (ii) this c-tuple match is bijective.*

EXAMPLE 2. *Using Q from Ex. 1 we generate the following three explanation pattern for $\langle d,6\rangle$.*

P_1 :	$R\langle d,6,true\rangle$	$Q(d,6)$:- $R(d,6), 6 > 5$
P_2 :	$S\langle d,6,N_1,N_1 < 5\rangle$	$Q(d,6)$:- $S(d,6,N_1), N_1 < 5$
P_3 :	$R\langle d,N_2,true\rangle, S\langle d,6,N_3,true\rangle$	$Q(d,6)$:- $R(d,N_2), S(d,6,N_3)$

P_1 matches ψ_1, because the single c-tuple in ψ_1 matches the single c-tuple of P_1. Similarly, P_2 matches ψ_2, and P_3 matches ψ_3 and ψ_4. On the other hand, ψ_5 does not match any of the explanation patterns and is not a valid explanation.

Ex. 2 shows that missing tuples can be generated from multiple explanation patterns P. We call the set of all explanation patterns that can generate all our missing tuples a *generic witness*.

DEFINITION 7 (GENERIC WITNESS). *Given a debugging scenario $\mathcal{S}$, a generic witness $W = \{P_1, P_2, \ldots, P_k\}$ is a set of explanation patterns, such that any explanation ψ for E matches at least one explanation pattern $P \in W$.*

4.2 Algorithm Overview

Our algorithm that produces an explanation set for SPJU queries consists of the following steps:

Step 1: Compute generic witness. Using $\mathcal{Q}$ and E, we first compute the *generic witness* W. Then, for each pattern P in the generic witness, do the following three steps:

- **Step 2: Create c-tables for D.** Convert all tuples in D into c-tuples with their condition set to *true* and insert them into c-tables. Add the c-tuples in the generic witness pattern P into the c-tables. We denote this set of c-tables derived from D as D_c. Ex. 6 shows a sample result of this step.
- **Step 3: Execute $\mathcal{Q}$ over the c-tables.** Compute $\mathcal{Q}(D_c)$ based on the semantics of query execution over c-tables defined in [13]. The resulting instances have the same schema as the instances obtained over D, except for an additional condition attached to each tuple. In these views, distinguish three types of tuples: (i) tuples that are present in the original view, (ii) tuples that match c-tuples in E, and (iii) the remaining tuples, which are potential side-effects. Ex. 7 shows examples of computed views, and Ex. 8 illustrates the identification of tuples of type (ii).
- **Step 4: Compute explanations.** Find every possible way of combining tuples of type (ii) such that each of the n c-tuples in E is generated exactly once. For each combination, a condition $C_i = t_1.cond \wedge \ldots \wedge t_n.cond$ is formulated as the conjunction of the conditions in each c-tuple. From the point of view of this current combination, all other tuples of type (ii) or (iii) are side-effects. If side-effects are not permitted or should be minimized (specified by $\mathcal{Q}_{im}$ or $\mathcal{Q}_{m}$,) we add conditions to C_i to enforce these conditions. Ex. 10 shows an example of a generated constraint. A *constraint solver* [10] is used to determine if C_i is satisfiable. If C_i is satisfiable, the current combination of c-tuples becomes an explanation.

Step 5: Prune and output explanations. Union the explanations returned for each pattern of the generic witness. This step further prunes redundant solutions if necessary.

We discuss the different steps of the algorithm next. Note that Step 5 is trivial and is not further discussed.

4.3 Generic Witness Generation (Step 1)

We first discuss how we obtain a generic witness (Def. 7) and then show how and under what conditions we minimize its size.
From queries to generic witnesses: Given an SPJU query Q and a missing tuple t, we compute a canonical instance. For unions of conjunctive queries, this canonical instance corresponds to a generic witness of t w.r.t. Q.

PROPOSITION 1. *Let Q be a union of m conjunctive queries of the form*

$$Q(Z_i) \text{ :- } R_{1_i}(X_{1_i}), \ldots, R_{k_i}(X_{k_i})$$

Let t be a valuation of x such that $Q(x)[\mathbf{x}/t]$, or $Q(t)$ for short is a tuple. $(E)[\mathbf{x}/t]$ means replace each variable $x \in \mathbf{x}$ with the corresponding value in t on the expression E. Then, the universal generic witness for Q includes m explanation patterns where the i-th pattern corresponds to the canonical instance of the i-th subquery, i.e., to

$$\left(R_{1_i}\left(X_{1_1}\right), \ldots, R_{k_i}\left(X_{k_i}\right)\right)[\mathbf{x}/t]$$

EXAMPLE 3. *Consider a query that is the union of the queries depicted in Fig. 3(a). Fig. 3(b) represents the universal generic witness for a tuple $t = \langle a,b \rangle$, computed as the canonical instance.*

Prop. 1 describes how to generate a universal generic witness for a single query Q and a single missing tuple t. However, our debugging scenario allows a set E of tuples missing from a set of query results $\mathcal{Q}(D)$. The semantics of E dictate that all tuples in E should occur simultaneously, so we need to determine a generic witness where each pattern generates all missing tuples. Such a generic witness is obtained by first determining the generic witness W_i for every missing tuple $t_i \in E$, $1 \leq i \leq n$. As a result, we create a set of patterns for each missing tuple. To obtain a set of patterns for all missing tuples, we simply combine patterns from the different witnesses using the cross-product:

$$W = W_1 \times W_2 \times \ldots \times W_n$$

For instance, if $W_1 = \{P_1, P_2\}$ and $W_2 = \{P_3, P_4\}$ then $W = \{P_1 \cup P_3, P_1 \cup P_4, P_2 \cup P_3, P_2 \cup P_4\}$.
Reducing the size of a generic witness: So far, our generic witness consists of a finite set of patterns that covers all possible explanations and thus satisfies Def. 7. In general, however, there is no guarantee that a generic witness is *minimal*. I.e., it is possible that all explanations matching a pattern $P_i \in W$ will also match a different patter $P_j \in W$. Obtaining a minimal generic witness is equivalent to determining if there exists an homomorphism from one pattern to another. (We are using the definition of homomorphism commonly used in data integration and data exchange [8].) It is also equivalent to solving query containment [1]. Thus, determining the minimal generic witness for unions of conjunctive queries is NP-complete. Moreover, query containment is undecidable when conjunctive queries contain inqualities (e.g., P_2).

Despite the computational complexity of this problem, minimizing the size of the generic witness is worth the effort since each pattern in the in the witness can generate numerous explanations. Whenever possible, we compute the core of each pattern [9]. Since cores of homomorphically equivalent patterns are equal (up to isomorphism), we use the core as unique representative of homomorphically equivalent patterns.

PROPOSITION 2. *Let t be a c-tuple with only constant values (no labeled nulls) and let Q be a union of conjunctive queries without inequalities. Let $P_i \in W$ be an explanation pattern of t w.r.t. Q. Then, $Core(P_i) = CoreWitness(P_i)$ and $W_{min} = \{CoreWitness(P_i) | P_i \in W\}$ is the unique minimum generic witness (up to isomorphism) for W.*

Notice that Prop. 2 applies when missing tuples contain only constant values. In general, missing tuples contain labeled nulls and in those cases the computed core may not be a valid pattern for an explanation (see Appendix E). In our implementation, we compute the minimum generic witness according to the following algorithm, when applicable.

PROPOSITION 3 (MINIMUM GENERIC WITNESS). *From the generic witness, we obtain a minimum generic witness by (i) computing the core of each pattern, (ii) removing duplicate patterns, and (iii) for each pair of remaining patterns (P_i, P_j) we remove P_j if $\exists h : P_i \rightarrow P_j$.*

EXAMPLE 4. *Consider the generic witness depicted in Fig. 3(b). We observe that it does not yet correspond to the minimum generic witness because pattern P_1 does not correspond to a core and $\exists h : P_1 \rightarrow P_3$ and $\exists h : P_2 \rightarrow P_3$. We compute the core of each pattern and obtain the result shown in Fig. 4(a). We then remove all patterns to which a homomorphism points to, e.g., P_3. The result is the minimum generic witness in Fig. 4(b).*

Even if the generic witness is minimal, all we guarantee is that we can use it to generate *all* possible explanations (and, thus, the

$Q(x,y) \text{ :- } R(x,y), R(x,z)$	$P_1 = R(a,b), R(a,N_1)$
$Q(x,z) \text{ :- } R(x,y), R(y,z)$	$P_2 = R(a,N_2), R(N_2,b)$
$Q(x,y) : -R(x,y), R(y,y)$	$P_3 = R(a,b), R(b,b)$
(a) query Q	(b) Generic witness

Figure 3: Computing a generic witness

$P_1 = R(a,b)$
$P_2 = R(a,N_2), R(N_2,b)$
$P_3 = R(a,b), R(b,b)$
(a) pattern cores

$P_1 = R(a,b)$
$P_2 = R(a,N_2), R(N_2,b)$
(b) minimal generic witness

Figure 4: Computing the minimal generic witness

generated explanation set is universal). However, the set of explanation generated is not necessarily minimal. Since it would be prohibitively expensive to compute homomorphism at the data instance level (i.e., among all explanations), we do not attempt this and pay the price of returning redundant explanations.

PROPOSITION 4 (COMPLETENESS & UNIVERSALITY). *The explanation set Ψ produced by Artemis for a (minimal) generic witness is complete and universal: Ψ is complete because all explanations matching a pattern of the generic witness are computed; Ψ is universal because there does not exist a valid explanation e' that is not covered by an explanation $e \in E$. An explanation e covers an explanation e' if every c-tuple in e matches a c-tuple in e' with equal labels(inserted (+) or existing).*

4.4 Conditional View Construction (Steps 2 and 3)

Conceptually, this step converts all source tables in D into a set of c-tables D_c. Then, we add the c-tuples in the patterns of the generic witness into D_c. Finally, we execute $\mathcal{Q}$ over the c-tables to obtain conditional views (c-views).

Converting source tables. For each relation $R \in D$, we create a c-table view definition $R'_c(X, true)$:-$R(X)$, i.e., we assign "true" as the condition of each c-tuple. In our implementation, all c-tables are views, thus avoiding any materialization.

Adding pattern c-tuples. We first create empty c-tables for each relation $R \in D$. That is, for each $R(X) \in D$, we create an empty c-table $\Delta R_c(X, cond)$. Then, for each $P \in W$ (the patterns in the generic witness), and for each c-tuple $t \in P$, we add t into its corresponding ΔR_c. The only important consideration here is what to do with the conditions in the c-tuple t. Conditions in c-tuples could involve values from the tuple alone (we call those *intra-tuple* conditions) or values from other tuples (*inter-tuple* conditions).

EXAMPLE 5. *Consider a pattern $P = \{R\langle N_1, 7, true\rangle, S\langle N_1, 3, N_3, N_3 > 5\rangle\}$ and the source tables of Ex. 1. Here $N_3 > 5$ is an intra-tuple condition. Notice, however that both tuples use N_1 to represent the same value. Let us rename the second N_1 and state the condition explicitly: $\{R\langle N_1, 7, N_1 = N_2\rangle, S\langle N_2, 3, N_3, N_1 = N_2 \wedge N_3 > 5\rangle\}$. $N_1 = N_2$ is an inter-tuple condition.*

When copying a c-tuple t into its corresponding ΔR_c, we only keep the intra-tuple conditions, because the inter-tuple conditions are introduced when generating c-views. We rename labeled nulls like N_1 in the previous example, but do not add inter-tuple conditions to ΔR_c. The actual R_c is the union of the c-tuples in R'_c and ΔR_c. We refer to these conditional source c-tables as D_c.

EXAMPLE 6. *The resulting c-tables for R and S for the previous example (D_c) are:*

R_c	A	B	*cond*
	a	1	*true*
	b	6	*true*
	d	3	*true*
	N_1	7	*true*

S_c	A	B	C	*cond*
	c	3	4	*true*
	a	9	7	*true*
	N_2	3	N_3	$N_3 > 5$

Note that if key or unique constraints exist on a relation R, we further add conditions to the c-tuples in ΔR_c that ensure that none of these tuples violates these constraints. For instance, assuming $R.A$ was a key in the above example, the c-tuple $\langle N_1, 7, true\rangle$ would change to $\langle N_1, 7, N_1 \notin \{a,b,d\}\rangle$.

C-view computation. We now execute $\mathcal{Q}(D_c)$ using the standard algorithm for query evaluation over c-tables [13]. We illustrate the intuition of this step with our example.

EXAMPLE 7. *Assume that $\mathcal{Q}$ consists of the following two queries and that R and S are as defined in the previous example.*

$$Q_1(B_1, C) \text{:-} R(A, B_1), S(A, B_2, C)$$
$$Q_2(B, C) \text{:-} S(A, B, C), C \neq 3$$

Applying these two queries on D_c, we obtain the following c-views.

Q_1	B	C	*cond*
	1	7	*true*
	7	4	$N_1 = c$
	7	7	$N_1 = a$
	1	N_3	$N_2 = a \wedge N_3 > 5$
	6	N_3	$N_2 = b \wedge N_3 > 5$
	3	N_3	$N_2 = d \wedge N_3 > 5$
	7	N_3	$N_1 = N_2 \wedge N_3 > 5$

Q_2	B	C	*cond*
	3	4	*true*
	9	7	*true*
	3	N_3	$N_3 > 5 \wedge N_3 \neq 3$

We implement the c-views as actual SQL queries on top of D_c. Notice that any potentially valid explanation must include at least one c-tuple form a ΔR_c part of an R_c relation. Otherwise, if all c-tuples come from the R'_c part of the relations, then the generated explanations will only include tuples that already exist in the source relations (i.e., the "missing" tuple cannot be missing). Thus, the query we generate uses the standard incremental view maintenance [4] pattern and treats the ΔR_c parts as the increments. I.e., given $Q = R \bowtie S$, we can compute new tuples in $Q(D_c)$ as $\Delta Q(D_c) = (\Delta R_c \bowtie S'_c) \cup (R'_c \bowtie \Delta S_c) \cup (\Delta R_c \bowtie \Delta S_c)$ Fig. 8 (Appendix) shows the c-view definition query for Q_1.

Missing-Answers [12] computes a query that in principle computes $Q(D \cup P)$, where P is the unique pattern of the canonical instance based on the single t and SPJ query Q the algorithm considers. The result of this query is the set of explanations returned to a user. We have identified several cases where the set of explanations can contain unsatisfiable explanations when using [12] (see Appendix C for details) and experiments show that the number of these incorrect explanations can be substantial. Our algorithm does not suffer from this problem because, as discussed next, it uses a constraint solver to distinguish between correct and incorrect explanations in addition to further considering side-effects.

4.5 Explanation Generation (Step 4)

In this final phase of explanation generation, we analyze the conditional views to generate all alternative explanations. We also consider the user-specified constraints on admissible side-effects ($\mathcal{Q}_{im}$ and $\mathcal{Q}_m$) and ensure that no explanation violates key or unique constraints. The explanation generation phase consists of two steps: (i) the identification of matching tuples and (ii) constraint formulation and execution.

Identifying matching tuples. For every missing tuple $t \in E$, we find matches in the corresponding $Q(D_c)$ computed in the previous step (recall that each missing tuple is associated with one query in $\mathcal{Q}$). Intuitively, t matches a c-tuple if the constant values match, the labeled nulls are unified, and the condition is satisfied. We denote the set of tuples matching t as $Match(t)$.

EXAMPLE 8. *Let us assume a tuple $t = \langle 7, 7\rangle \in E$ is missing from $Q_1(D)$ (from the previous example). We scan the c-tuples in $Q_1(D_c)$ and determine $Match(t) = \{\langle 7, 7, N_1 = a\rangle, \langle 7, N_3, N_1 = N_2 \wedge N_3 > 5\rangle\}$.*

Resolving constraints. In this second step we use a *constraint solver* to determine which tuples in $Match(t)$ satisfy all constraints in the query, t, $\mathcal{Q}_m$ and $\mathcal{Q}_{im}$. Every tuple $t_m \in Match(t)$ that the constraint solver determines to be satisfiable becomes an explanation for the tuple t.

In our implementation, we use MINION [10] as a constraint solver. To solve a constraint, MINION requires the following information: (i) the specification of attribute domains, (ii) variable declarations, including the variable name and variable domain, and (iii) the constraint itself. We create these declarations for each debugging scenario $\mathcal{S}$ and initialize MINION with them. Fig. 10 in Appendix B.2 shows an example input file for MINION.

Every variable that appears in a constraint must be assigned a domain. The main data type in MINION are integers and we need to covert some of our data types into integers before using the solver. For instance, we have a function $mapInt()$ that maps strings into an integer representation. Since we have to handle comparisons of string values in our queries, $mapInt()$ makes sure the mapping preserves the lexicographical order of the string values. This mapping is performed per debugging scenario and the domain definitions are reused for every tuple $t_m \in Match(t)$. To model finite domains for key or unique attribute values, we exclude all existing key values already in the attribute.

EXAMPLE 9. *For the variables N_1, N_2, and N_3 of our example, assume that R.A is a key attribute. Then, the domain definition for the attribute of N_1, i.e., R.A, is $dom(R.A) = \mathbb{Z} \setminus \{mapInt(\mathsf{a}), mapInt(\mathsf{b}), mapInt(\mathsf{d})\}$, whereas the remaining domain definitions are simply $\mathbb{Z}$.*

Note that the exclusion of key and unique values from the domains allows us to remove the constraint that $N_1 \notin \{a,b,d\}$ from the constraints of c-tuples in the c-view (as it is enforced by the definition of the domain). However, this optimization only applies for key and unique constraints over a single attribute. To enforce composite keys, the corresponding constraints remain in the c-tuples and are passed to the constraint solver.

Variables are declared by simply defining their name and binding them to one of the domains. For instance, the domain of N_1 is declared to be $dom(R.A)$.

By default, the condition we send to the constraint solver for each t_m is its $t_m.cond$ condition. We now discuss how we modify this default constraint to deal with side-effects.

A side-effect for a tuple t_m is any tuple $t_{se} \in \Delta Q(D_c)$ s.t. (i) $t_{se} \neq t_m$ and (ii) the existence of t_m implies the existence of t_{se}. Let us denote the set of potential side-effects as $SE(t_m) = \{t_{se_1}, t_{se_2} \dots, t_{se_n}\}$ which we obtain by executing the query $\Delta Q(D_c) \setminus \{t_m\}$.

In the case where no side-effects are admissible, i.e., when $Q \in \mathcal{Q}_{im}$, we send the following constraint C_{im} to the constraint solver, which is satisfied iff the condition of t_m is true and none of the potential side-effect tuples exist, i.e., all conditions of potential side-effects evaluate to false.

$$C_{im} = t_m.cond \wedge \neg\,(t_{se_1}.cond \vee t_{se_2}.cond, \dots \vee t_{se_n}.cond)$$

When side-effects are allowed, but must be minimized, we generate a constraint C_m that asserts a boolean variable for every side-effect. We then add to the constraint the requirement that the number of these boolean variables asserted as true to be minimal. To understand the expression, assume each variable is 1 when true and 0 when false.

$$\begin{array}{rcl} C_m & = & t_m.cond \\ & \wedge & (t_{se_1}.cond \Leftrightarrow i_1) \wedge (t_{se_2}.cond \Leftrightarrow i_2) \dots \wedge (t_{se_n}.cond \Leftrightarrow i_n) \\ & \wedge & min(i_1 + i_2 + \dots + i_n) \end{array}$$

EXAMPLE 10. *Assume $t_m = \langle 7, N_3, N_1 = N_2 \wedge N_3 > 5\rangle$ and that Q_1 is immutable. In this case, we pass the following constraint to a constraint solver:*

$$\begin{array}{l} (N_1 = N_2) \wedge (N_3 > 5) \wedge \\ \neg((N_1 = c) \vee (N_1 = a) \vee (N_2 = a \wedge N_3 > 5) \vee (N_2 = b \wedge N_3 > 5) \vee (N_2 = d \wedge N_3 > 5)) \end{array}$$

In our implementation, we reduce the set of potential side-effects based on the concept of insertion patterns. Intuitively, given a matching tuple t_m, we only consider as potential side-effects those tuples in the c-view that insert c-tuples to the same set of tables as required for t_m, or a subset thereof. For instance, assume $t_m = \langle 7, 7, N_1 = a\rangle$. The explanation of this tuple only inserts a c-tuple to R and we thus identify a potential side-effect as any c-tuple in the view that only inserts data to R as well. We observe that only the second tuple of the c-view in our example also inserts to R only, so the constraint checking for side-effects for match t_m consists of conditions contributed by one instead of five c-tuples.

The number of constraints passed to the constraint solver is bound by $|Match(t_1)| + |Match(t_2)| + \dots + |Match(t_n)|$ for $E = \{t_1, t_2, \dots t_n\}$. In the worst case, this number grows exponentially with the number of joins in the query and the computational complexity of the Artemis algorithm is dominated by the complexity of performing this step. In practice, the number of constraints we pass to the solver is reduced because the explanations we produce for each $t_m \in Match(t_i)$ are sorted by the descending number of inserts required. That is, an explanation t_m that requires only one insert will appear at the end of the produced set. Intuitively, if an explanation t_m that requires k inserts is not satisfiable, any t'_m that contains a subset of the inserts in t_m is not satisfiable as well.

The size of the constraint, measured by the number of c-tuples that contribute to the condition, depends on the constraints imposed by the debugging scenario. In the simplest case, when there is no need to consider side-effects and there are no key or unique constraints, the solver only needs to deal with the conditions in $t_m.cond$. On the other hand, when side-effects are considered, the number of c-tuples involved per constraint is bounded by $|\Delta\mathcal{Q}(D_c)|$. However, it can be reduced to $\Delta\mathcal{Q}(D'_c \cup \Delta t_m)$ for a given matching tuple t_m, where Δt_m includes the insertions of c-tuples relevant to produce t_m, which is given by the insertion pattern corresponding to t_m.

5. Grouping & Aggregation

We now briefly discuss some modifications to our SPJU algorithm to handle grouping and aggregation. Consider, for example, the following query and its result:

```
SELECT P.PID, P.Picture, CCOUNT(UI.UID) AS C
FROM Picture P, PictureTag PT, UserInterest UI
WHERE P.PID = PT.PID AND UI.IID = PT.Category
  AND P.Visibility = 'Public' AND PT.Category = 'I4'
GROUP BY P.PID, P.Picture
```

$Q(D)$	PID	Picture	C
	P1	pier39.jpg	2

Our goal is to use the algorithm described in Sec. 4 to compute explanations for the SPJ parts of the query and then process those explanations to handle grouping and aggregation. We handle grouping by grouping the resulting explanations for the SPJ part by the value of the group by columns (in effect, creating groups of explanations over which the aggregate function is applied). However, we leave it to the user to decide which subset of explanations in each group is needed to properly create the aggregated value. We explain this with an example.

EXAMPLE 11. *Assume we want to explain why the tuple* $\langle \text{P4}, \text{winetasting.jpg}, 18\rangle$ *is not in* $Q(D)$ *above. Artemis produces the following explanation* ψ*:*

$$\begin{array}{lll} [& \{P\langle \text{P4}, \text{winetasting.jpg}, \text{Public}\rangle, +PT\langle v_{ptid}, \text{P4}, \text{I4}\rangle\}, +UI\langle \text{I4}, v_{uid}\rangle & (1) \\ & \{P\langle \text{P4}, \text{winetasting.jpg}, \text{Public}\rangle, +PT\langle v_{ptid}, \text{P4}, \text{I4}\rangle\}, UI\langle \text{I4}, \text{U1}\rangle & (2) \\ & \{P\langle \text{P4}, \text{winetasting.jpg}, \text{Public}\rangle, +PT\langle v_{ptid}, \text{P4}, \text{I4}\rangle\}, UI\langle \text{I4}, \text{U3}\rangle & (3) \\ ] & \text{s.t. } groupBy(\text{P.PID = P4, P.Picture = `winetasting.jpg'}) \wedge count(\text{UI.UID}) = 18 & \end{array}$$

(We are using the relation aliases instead of the names in this example). To read this explanation, start from the group by constraint at the bottom. We can generate the missing tuple for the group where P.PID is P4 and P.Picture is 'winetasting.jpg' and where the total count is 18 if we insert a number of tuples in the source relations that match any one of the patterns (labeled (1), (2), and (3)). For example, we can insert 9 tuples into PictureTag with PID = 'P4' and IID = 'I4'. Since these 9 new tuples (in combination with the existing Picture and UserInterest tuple) will match patterns (2) and (3), the missing tuple with a count of 18 is generated. Alternatively, we can insert a tuple in PictureTag and UsetInterest that, together, satisfy pattern (1) and five more PictureTag tuples that, again, match both (2) and (3). Notice that these explanations only provide the pattern for the missing tuples. However, it is still up to the user to select the appropriate combination of tuples that satisfy the explanation patterns and that generate the correct aggregate value in the missing tuple.

To find the canonical instances of an SPJUA query, the SPJUA explanation algorithm starts by dividing the input query into its SPJA sub-queries. Then, selections, projections, and joins are pushed down the aggregations. That is, we divide each SPJA into an SPJ block, a set A of aggregate attributes, and a set G of grouping attributes. This procedure is similar to computing the canonical instance in the presence of aggregation introduced in [7]. In our current example, Q does not have a UNION clause and, thus, Q is already an SPJA query. The SPJ block of Q is

```
SELECT P.PID, P.Picture, UI.UID
FROM Picture P, PictureTag PT, UserInterest UI
WHERE P.PID = PT.PID AND UI.IID = PT.Category
  AND P.Visibility = 'Public AND PT.Category = 'I4'
```

We compute A and G from the SELECT and GROUP BY clauses of the SPJA components. In this example, $A = \{UI.UID\}$ and $G = \{P.PID, P.Picture\}$.

We then convert each missing tuple $t \in E$ into a tuple t' that replaces all aggregate attributes in t to labeled nulls in t'. More formally, given a c-tuple $t \in E$, we create a c-tuple $t' = \langle \pi_G(t), Var(A), Cond(t.cond, G)\rangle$, where $Var(A)$ generates a new labeled null for every attribute in A, and $Cond(cond, G)$ drops any condition in the original $t.cond$ that does not use the attributes in G. We do not support conditions on the aggregated values yet other than the actual resulting aggregate values. Since it is up to the user to put together a valid combination of tuples that satisfies the aggregate condition, conditions on the aggregated values are handled by users. This is a limitation we plan to address in future work. In the case of our example, $t = \langle \text{P4}, \text{winetasting.jpg}, 18\rangle$ is converted into $t' = \langle \text{P4}, \text{winetasting.jpg}, v_1\rangle$.

Given an SPJA query Q with grouped attributes G and aggregated attributes A, the explanation of a missing tuple t is given by a single explanation of the form

$$\Psi_{SPJA} : [\Psi] \text{ s.t. } groupBy(G) \wedge agg(A) = \pi_A(t)$$

where Ψ is the universal explanation set (see Def. 4) of the SPJ block of Q.

When Q is a union of SPJA queries, we proceed analogously to the SPJU case, which results in one SPJA-explanation for every sub-query. The set of SPJA-explanations for one SPJAU query is denoted as Ψ_{SPJAU}. In the general case, E consists of more than one tuple. Let $\Psi_{i_{SPJUA}}$ be the explanation for tuple $t_i \in E, 1 \le i \le n$, then, the explanation of E is given by

$$\Psi_{1_{SPJAU}} \times \Psi_{2_{SPJAU}}, \ldots \times \Psi_{n_{SPJAU}}$$

6. Evaluation

We implemented both Artemis and the *Missing-Answers* system in [12] using Java 1.6. The experiments reported in this section were run on a Mac Book Pro with a 2.93 GHz Intel Core Duo processor, 4GB of main memory, and running Mac OS 10.6.2. We used IBM DB2 V9.5 as our DBMS and run it locally on the same machine, using the out-of-the-box parameter settings. Artemis requires the use of a constraint solver for validating the generated explanations. We used MINION v0.8.1 [10], a general-purpose and open-source constraint solver.

We used 10 MB of TPC-H data (corresponding to tens of thousands of tuples in some tables), a well-known query benchmark for decision support (http://www.tpc.org/tpch/). TPC-H defines a relational database that models parts, part suppliers, and orders for parts. The benchmark defines 22 queries of varying complexity over this relational data. Since some of these queries include features that Artemis does not support (e.g., HAVING clauses, nested queries), the queries we use are adaptations of some of the TPC-H queries. To compare Artemis with Missing-Answers, we only consider the SPJU parts of the queries, except for $Q1$.

TPC-H includes two tables that encode geographic location of suppliers and customers (tables *Nation* and *Region*). Since we consider the information in those tables to be authoritative, we do not want explanations that require inserting new tuples into them. Thus, $Q_{\not m} = \{Nation, Region\}$. Also, in the results reported here, we assume $Q_{\not n} = \emptyset$.

Finally, to compare with Missing-Answers, E contains a single missing tuple. Since the number of constraints we put into each missing tuple affects the runtime (and the number of explanation generated), the missing tuples we used for each query contain a mix of constant values and labeled nulls. Specifically, we put a constant value for any categorical field in the query schema (e.g., string and date fields) and we used labeled-nulls for all other fields (e.g., numerical fields). All time results are the average time for five runs of the same scenario.

Note that more details on the TPC-H scenario and a second scenario we used (PhotoShare) are available in Appendix D, together with further experiments.

Experiment 1: Explanation quality. Since Artemis uses a constraint solver to evaluate potential explanations, we guarantee all our solutions satisfy the constraints of the scenario. Missing-Answers does not have that guarantee and, indeed, returns solutions that do not satisfy some constraints (see Appendix C for details).

Fig. 5 compares the number of explanations Artemis and Missing-Answers return for a single missing tuple for a number of TPC-H queries. To understand how to read these results, consider the bar for Q_5. In that case, Artemis returns 451 explanations, all satisfying the constraints of the scenario, while Missing-Answers returns 19,254 explanations. The extra 18,803 explanations returned by Missing-Answers do not satisfy the constraints of this scenario and are considered wrong. For Q_5, the main reason of the overhead explanations generated by Missing-Answers is the presence of inequalities in the query that may cause wrong explanations. Another reason for instance applies to Q_3, where we specify a missing tuple

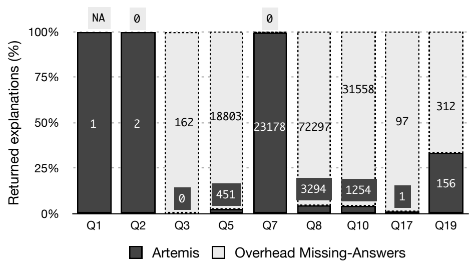


Figure 5: Result quality on TPCH data

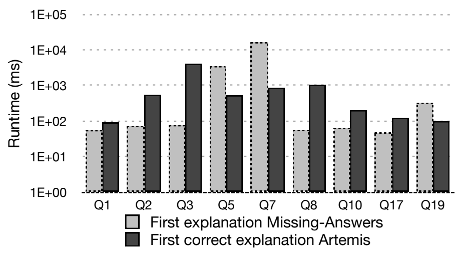


Figure 6: Result quality on TPC-H data

that is unsatisfiable based on the query and we observe that Artemis correctly returns no explanations while Missing-Answers generates 162 explanations.

We conclude that the use of a constraint solver significantly reduces the number of explanations generated in many scenarios. The key question we answer next is how much more (or less) time we use to generate explanations using Artemis.

Experiment 2: Runtime to compute first correct explanation. Fig. 6 shows how long developers have to wait for the **first** explanation to be returned when using Artemis and Missing-Answers. By definition, all explanations returned by Artemis satisfy all constraints and, thus, the first explanation is considered correct. (Notice that this is different from being the solution the user wants.) Missing-Answers, however, may return a number of wrong explanations before producing the first correct one. Nevertheless, for the purpose of runtime comparison, we ignore this fact here.

We observe that in eight out of nine debugging scenarios, Artemis returns the first explanation in less than a second, which indicates that a developer does not have to wait a significant time before starting to process explanations. For the unsatisfiable query Q_3, the runtime of Artemis is identical to the runtime of computing all explanations, because it is a query where no explanation exist and all constraints are thus processed. For that query, Missing-Answers produces 162 explanations so the extra time needed to verify that none of these are correct explanations is worthwhile.

Comparing the runtimes of Artemis and Missing-Answers, we observe that Artemis returns the first explanation faster than Missing-Answers in three debugging scenarios. In all other cases, using the constraint solver to verify the correctness of explanations adds an overhead to the runtime, but, as mentioned previously, this results in a higher precision of the returned set of explanations that saves a developer to manually go through (wrong) explanations.

7. Conclusion and Outlook

We presented an algorithm that determines instance-based explanations, an idea originally introduced in [12]. This algorithm computes all instance-based explanations of missing answers for a set of SPJUA queries and implements the framework for instance-based explanations we introduced in this paper. Our algorithm can answer "why-not" questions for multiple missing tuples at once and allows users to embed conditions in those "why-not" question. At the heart of our algorithm is the use of a constraint solver to ensure all explanations are satisfiable with respect to the underlying database constraints and requirements expressed by users. Our framework and algorithms also consider side-effects created by the explanations and allow users to either avoid or minimize them. We further studied interesting properties such as universality and minimality of the the set of explanations returned by our algorithm. Experiments suggest that the runtime is acceptable to effectively use the resulting explanations for query analysis. Nevertheless, we plan to investigate further methods to reduce the runtime.

Besides instance-based explanations, missing tuples can also be explained based on queries [5]. The goal of Artemis is to combine the best of query-based and instance-based explanations into one query debugging tool. Besides generating correct explanations, it is also worthwhile to study how to easily convey their meaning to users, for instance based on suited visualization techniques or presentation as natural language instead of c-tuples.

Acknowledgements. We thank Prof. Wang-Chiew Tan (UCSC) for the advice and fruitful discussions that helped define our framework and algorithms.

8. References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] A. Atserias, A. Dawar, and P. G. Kolaitis. On preservation under homomorphisms and unions of conjunctive queries. *Journal od the ACM*, 53(2):208–237, 2006.

[3] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *International Conference on Data Engineering (ICDE)*, pages 506–515, Istanbul, Turkey, 2007.

[4] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Conference on the Management of Data (SIGMOD)*, pages 61–71, 1986.

[5] A. Chapman and H. V. Jagadish. Why not? In *SIGMOD Conference*, pages 523–534, 2009.

[6] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

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

[8] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[9] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. *ACM Transactions on Database Systems*, 30(1):174–210, 2005.

[10] I. P. Gent, C. Jefferson, and I. Miguel. Minion: A Fast Scalable Constraint Solver. In *ECAI*, pages 98–102, 2006.

[11] M. Herschel, M. A. Hernández, and W. C. Tan. Artemis: A system for analyzing missing answers. *PVLDB*, 2(2):1550–1553, 2009.

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

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

[14] J. Lechtenbörger, H. Shu, and G. Vossen. Aggregate queries over conditional tables. *J. Intell. Inf. Syst.*, 19(3):343–362, 2002.

[15] A. Meliou, W. Gatterbauer, K. Moore, and D. Suciu. Why so? or why no? functional causality for explaining query answers. Technical Report 09-12-01, University of Washington, 2009.

[16] R. Ramakrishnan. *Database Management Systems*. WCB/McGraw-Hill, 1998.

[17] H. Shu. View maintenance using conditional tables. In *Deductive and Object-Oriented Databases (DOOD)*, pages 67–84, 1997.

[18] H. Shu. Using constraint satisfaction for view update. *Journal on Intelligent Information Systems*, 15(2):147–173, 2000.

APPENDIX

A. PhotoShare Sample Queries

In Sec. 1, we presented the PhotoShare scenario and showed sample data for a selection of tables and views in Fig. 1. Fig. 7 shows the SQL queries (both SPJU queries) that define the views *Network* and *InterestingPics*.

```
CREATE VIEW Photoshare.Network AS
SELECT  U1.email AS U1_email,
            U2.name AS Friend, U2.email AS U2_email
FROM    Photoshare.User U1, Photoshare.User U2,
            Photoshare.Friend F
WHERE   U1.uid = F.uid1 and u2.uid = f.uid2
UNION
SELECT  U1.email AS U1_email,
            U2.name AS Friend, U2.email AS U2_email
FROM    Photoshare.User U1, Photoshare.User U2,
            Photoshare.Friend F
WHERE   U1.uid = F.uid2 and u2.uid = f.uid1;

CREATE VIEW Photoshare.InterestingPics AS
SELECT  U1.EMAIL AS U1_EMAIL, P.Picture AS Picture,
            U2.Name AS PContributor
FROM    Photoshare.User U1, Photoshare.User U2,
            Photoshare.PICTURE P, Photoshare.PictureTag PT,
            Photoshare.UserInterest UI
WHERE   U1.UID = UI.UID AND PT.CATEGORY = UI.IID
AND         PT.PID = P.PID AND P.VISIBILITY = 'Public'
AND         U1.UID <> U2.UID AND U2.UID = P.UID
UNION
SELECT  NWV.U1_EMAIL AS U1_EMAIL, P.PICTURE AS Picture,
            NWV.FRIEND AS PContributor
FROM    Photoshare.User U1, Photoshare.User U2,
            Photoshare.PICTURE P, Photoshare.PictureTag PT,
            Photoshare.UserInterest UI, Photoshare.Network NWV
WHERE   NWV.U1_EMAIL = U1.EMAIL AND UI.UID = U1.UID
AND         NWV.U2_EMAIL = U2.EMAIL AND U2.UID = P.UID
AND         PT.PID = P.PID AND PT.CATEGORY = UI.IID
AND         P.VISIBILITY = 'Friend'
```

Figure 7: View definition queries for Fig 1

B. Explanation Generation Details

B.1 C-View Query Generation (Step 3)

In Sec. 4.4 we discussed how we construct a *conditional view* for the query Q that creates explanations for a missing tuple in that query. The main intuition of this part of the algorithm is that we treat the c-tuples corresponding to the explanation patterns in the generic witness as *incremental* portions of the source tables. Since we are only interested in solutions that use at least one of these explanation patterns, the query follows the familiar pattern from incremental view maintenance.

Fig. 8 shows the SQL query used to compute this incremental c-view ($\Delta Q_1(D_c)$) for the query Q_1 in Ex. 7 (`SELECT R.B, S.C FROM R, S WHERE R.A=S.A`). The query assumes the user wants to know why the tuple $\langle 7,7 \rangle$ is missing from the answer of Q_1. The `VALUES` clauses in the query add the generic witness patterns into the resulting explanations.

Fig. 9 shows the result computed by the SQL query in Fig. 8 over the data in Ex. 6. Note that the `INSERT` columns keep track if the source tuples exist (`F`) or must be inserted (`T`) and define the insertion pattern of an explanation.

B.2 Explanation Generation (Step 4)

Explanation generation iterates over each tuple in the c-view (Fig. 9) and first identifies the tuples matching the missing tuple. Assuming that $\langle 7,7 \rangle$ is the missing tuple, the first row is a match that describes a combination of the existing tuple $S\langle a,9,7 \rangle$ and an inserted tuple $+R\langle v3,7,v3 = a \wedge a = v3 \rangle$.

```
SELECT -- Delta S Join Delta R
  RTRIM(CHAR(S.A)) AS S_A, RTRIM(CHAR(S.B)) AS S_B,
  RTRIM(CHAR(S.C)) AS S_C,
  RTRIM(CHAR(R.A)) AS R_A, RTRIM(CHAR(R.B)) AS R_B,
  RTRIM( coalesce(CHAR( R.A), 'null') ) || '='
          || RTRIM( coalesce(CHAR( S.A ), 'null') )||
   '/\' || RTRIM( coalesce(CHAR( S.A), 'null') ) || '='
          || RTRIM( coalesce(CHAR( R.A ), 'null') )
  AS Condition,
  'T' AS INSERT_S, 'T' AS INSERT_R
FROM
  ( VALUES ( 'v1', 'v2', '7') ) AS S(A, B, C),
  ( VALUES ( 'v3', '7') ) AS R(A, B)
UNION
SELECT -- Delta S Join R
  RTRIM(CHAR(S.A)) AS S_A, RTRIM(CHAR(S.B)) AS S_B,
  RTRIM(CHAR(S.C)) AS S_C,
  RTRIM(CHAR(R.A)) AS R_A, RTRIM(CHAR(R.B)) AS R_B,
  RTRIM( coalesce(CHAR( R.A), 'null') ) || '='
             || RTRIM( coalesce(CHAR( S.A ), 'null') )||
      '/\' ||RTRIM( coalesce(CHAR( S.A), 'null') ) || '='
             || RTRIM( coalesce(CHAR( R.A ), 'null') )
  AS Condition,
  'T' AS INSERT_S, 'F' AS INSERT_R
FROM
  ( VALUES ( 'v1', 'v2', '7') ) AS S(A, B, C),
  R AS R
UNION       -- S Join Delta R
SELECT
  RTRIM(CHAR(S.A)) AS S_A, RTRIM(CHAR(S.B)) AS S_B,
  RTRIM(CHAR(S.C)) AS S_C,
  RTRIM(CHAR(R.A)) AS R_A, RTRIM(CHAR(R.B)) AS R_B,
  RTRIM( coalesce(CHAR( R.A), 'null') ) || '='
            || RTRIM( coalesce(CHAR( S.A ), 'null') )||
      '/\' ||RTRIM( coalesce(CHAR( S.A), 'null') ) || '='
      || RTRIM( coalesce(CHAR( R.A ), 'null') )
  AS Condition,
  'F' AS INSERT_S, 'T' AS INSERT_R
FROM
  S AS S,
  ( VALUES ( 'v3', '7') ) AS R(A, B)
```

Figure 8: C-view implementation for Q_1

S_A	S_B	S_C	R_A	R_B	CONDITION	INSERT_S	INSERT_R
a	9	7	v3	7	v3=a∧a=v3	F	T
c	3	4	v3	7	v3=c∧c=v3	F	T
v1	v2	7	a	1	a=v1∧v1=a	T	F
v1	v2	7	b	6	b=v1∧v1=b	T	F
v1	v2	7	d	3	d=v1∧v1=d	T	F
v1	v2	7	v3	7	v3=v1∧v1=v3	T	T

Figure 9: Result of c-view query of Fig. 8

For each matching tuple, a logical constraint is formulated that, if satisfied, would make the tuple a valid explanation for the debugging scenario. As we discussed in Sec. 4.5, these constraints are used to (i) validate the condition of each c-tuple, (ii) make sure the explanation does not violate a key constraint, and (iii) avoid the creation of side-effects when needed. The constraints are passed to MINION, a constraint-solver, that decides if they are satisfiable.

We use MINION through Essence', a declarative language for specifying constraints[1] and that serves as a front-end to MINION. An Essence' input declaration consists of a header, domain definitions, variable declarations, and the constraint itself. Fig. 10 shows an example input to MINION written in Essence'. To generate the domain declarations, we have to convert every infinite domain to an order preserving finite integer domain. To this end, we map each constant that exists in D and E for a given attribute, e.g., $R.A$ to an integer such that the order of integers preserves the order of strings. In our example, $a \longrightarrow 1$, $b \longrightarrow 2$, and $d \longrightarrow 3$. We further extend the domain by integers at the head and tail of the domain of constants by a number of values that can stand in for the labeled nulls, based on the number of labeled nulls and comparisons performed over these. For instance, the value -1 stands for any labeled null with value less than another value, whereas 4 stands for any labeled null larger than any other value.

[1]http://www.cs.st-andrews.ac.uk/ andrea/tailor/

```
language ESSENCE' 1.b.a
$$ attribute domains
letting r_r_a be domain int {-1, 1, 2, 3, 4}
letting SEC be domain int
$$ Variable declarations (domains excluded for brevity)
find v3:r_r_a
find i1:bool
find s:SEC
$$ Constraint to minimize number of side-effects
minimising s
such that
    $$ condition of matching tuple t_m
    ( (1=v3) )
    $$ Conditions of potential side-effects
    ∧( ( 3=v3) )<=> (i1) )
    $$ Counting side-effect
    ∧ (s = ( i1 ) )
```

Figure 10: Constraint input to MINION in Essence'

C. Analysis of Missing-Answers Algorithm

At a very high-level, Missing-Answers transforms the input query Q into a new query Q' that computes a canonical instance for the "missing" tuple t. The modified query returns existing tuples from the underlying tables whose values match those in t combined with *dummy* tuples that stand for missing values on the source. Each table in the FROM clause of Q will receive one extra dummy tuple containing null values for all attributes. The WHERE clause in Q' contains selection conditions that (i) select the attribute values in t and (ii) make sure that if the values in t do not exists in the source table, then the dummy tuple is picked as a place-holder.

For example, for query Q_1 of Ex. 7 and assuming the missing tuple $t = \langle 7,7 \rangle$, Missing-Answers produces the query shown in Fig. 11, whose result corresponds to the set of explanations. The `VALUES` clauses add the dummy tuples and the additional disjunctions on the `WHERE` clause select the dummy tuples when the missing values are not available in the source.

```
SELECT ...
FROM
    (SELECT R.A, R.B FROM R UNION
     VALUES (NULL, NULL)
    ) R,
    (SELECT S.A, S.B, S.C FROM S UNION
     VALUES (NULL, NULL, NULL)
    ) S
WHERE (R.A = S.A OR R.A = NULL OR S.A = NULL)
    AND (R.B = 7 OR R.B = NULL)
    AND (S.C = 7 OR R.C = NULL)
```

Figure 11: SQL query produced by Missing-Answers

We now discuss three cases where Missing-Answers produces explanations that cannot be considered correct given the constraints in our debugging scenarios.

Dealing with null values. Missing data from the sources is modeled as tuples that consist of null values only. Predicates in the WHERE clause are then modified to ensure that an attribute either satisfies the constraints of Q and t based on its attribute value, or the attribute value matches the value of a missing tuple, i.e., NULL. Clearly, if the source data includes NULL values in these attributes, incorrect explanations may be generated.

Inequalities. Missing-Answers produces correct explanations in the restricted case where all conditions in the WHERE clause are equality conditions (and none of the other cases discussed here apply). However, when inequalities appear in the `WHERE` clause, Missing-Answers can produce unsatisfiable explanations. As a very simple example, consider the query `SELECT B FROM R WHERE B > 3` and assume that the missing tuple is $t = \langle 2 \rangle$. The generated query follows this general pattern:

```
SELECT ...
FROM (SELECT * FROM R UNION VALUES(NULL, NULL)) R
WHERE ( R.B = 2 OR R.B IS NULL )
  AND ( R.B > 3 OR R.B IS NULL )
```

The query returns the dummy tuple for R with the constraint that $R.B = 2$ and under the unsatisfiable condition $2 > 3$. Since Missing-Answers does not check the condition, the returned tuple is considered an explanation.

Key and foreign key constraints. The way the query generation algorithm of Missing-Answers is defined, the rewriting that considers key and unique constraints may still return explanations that violate these constraints. For instance, assuming attribute $R.A$ in Ex. 1 is a key attribute, the query generated by Missing-Answers is the same as the query of Fig. 11, except that a full outer join combines the two tables R and S that both, however, still contain a dummy tuple. The query returns one explanation that violates the constraint, i.e.,$+R\langle N_1, 7, N_1 = a \rangle$,$S\langle \mathsf{a}, 9, 7 \rangle$

Note that there are no trivial extensions of Missing-Answers that would address the above.

D. Detailed Experimental Results

We report results for debugging scenarios that originate from one of the following domains.

- **PhotoShare** corresponds to the domain described in the example of Sec. 1. We defined six debugging scenarios over the base tables (the tables of Fig. 1 plus tables *Interest(IID, Name)* and *PictureVote(PID, UID, Vote)*). Fig. 12 summarizes the debugging scenarios. The data set itself represents a small database a developer may generate to study the behavior of queries (tens of tuples to a hundred tuples). Such a small amount of data allows us to manually analyze the set of explanations returned by each algorithm. When not mentioned otherwise, $Q_{im} = Q_m = \emptyset$, and $\mathcal{Q}$ consists of a single query only.
- **TPC-H.** We described the TPC-H scenario in Sec. 6 and summarize additional relevant information on the debugging scenarios in Fig. 13. Note that the Q_1 scenario uses aggregation and the missing tuple asks about the aggregated value. In the Q_3 scenario we used a missing tuple with values for which we know there is no possible valid explanation.

Experiment A1: Result quality on PhotoShare. For PhotoShare, we process the six debugging scenarios using both Artemis and Missing-Answers, when applicable. We measure the number of explanations returned for each debugging scenario by each algorithm and report them in Fig. 14. We manually verify the set of explanations returned by each algorithm w.r.t. the correctness of an explanation. Whereas Artemis returns correct explanations only, Missing-Answers returns additional incorrect explanations.

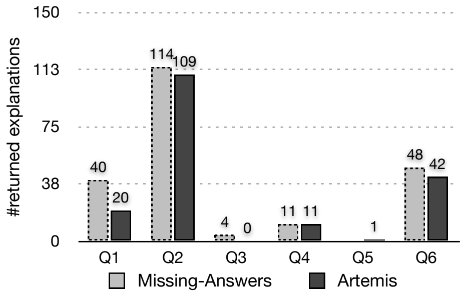


Figure 14: Result quality on PhotoShare data

As for the TPC-H debugging scenarios (see Exp. 1 in Sec. 6), we observe that Missing-Answers often returns incorrect explanations.

Query	View Schema	Missing tuple t
Q_1	UserEmailInterest(UEmail, Interest)	$\langle \$email, \$interest \rangle$
Q_2	InterestingPics(U1_Email, Picture, PContributor)	$\langle$john@univ.edu, winetasting.jpg, Jane$\rangle$
Q_3	UnSat(UID, PID)	$\langle 3, 2 \rangle$
Q_4	UserNameInterest(UName, Interest)	$\langle$Jane, Bridge$\rangle$
Q_5	PopularPublicPics(Pict, Contrib, AvgVote)	$\langle$sausalito, $\$contrib$, 5$\rangle$
Q_6	Network(U1_Email, Friend, U2_Email)	$\langle \$email1, \$name, \$email2 \rangle$

Figure 12: View schemas and missing tuples in PhotoShare debugging scenarios

Query	Type	Inequalities?	Key/Unique violation?
Q_1	SPJA	yes	no
Q_2	SPJ	yes	no
Q_3	SPJ	yes	no
Q_5	SPJ	yes	yes
Q_7	SPJU	yes	no
Q_8	SPJ	yes	yes
Q_{10}	SPJ	yes	no
Q_{17}	SPJ	yes	no
Q_{19}	SPJU	yes	yes

Figure 13: TPC-H debugging scenarios

We discussed in Appendix C that Missing-Answers can yield unsatisfiable explanations due to key and unique constraint violations (the case for Q_1, Q_2, and Q_6), inequalities (the case for Q_2 and Q_3), an null values (the case for Q_2).

Among the PhotoShare scenarios, Q_1 is a query that projects out a unique attribute (*User.Email*), as opposed to Q_4 that is the same query except for the projection. When bound to a constant in a missing tuple, tuples added to the *User* relation by an explanation have to make sure that the constraint is not violated. The additional explanations returned by Missing-Answers violate unique constraints. Similarly, Missing-Answers explanations violate key constraints in Q_4. However, in this case constants in explanations that cause this violation are not introduced during generic witness generation, but when combining existing tuples with new c-tuples during the c-view generation. Q_3 uses a debugging scenario where the missing tuple is unsatisfiable based on the join condition *UI.UID* < *PT.PID*. This case is not correctly handled by Missing-Answers, which returns 4 explanations in this unsatisfiable case. Finally, Missing-Answers does not apply to Q_5, as it is an SPJA query.

Experiment A2: Runtime to compute all explanations. We now study the runtime of Artemis to compute all explanations. Here, we report the runtime for the TPC-H debugging scenarios for both Artemis and Missing-Answers. For Missing-Answers, all runtimes except for Q_5 and Q_7 are below 1 second. Q_5 takes 3.5 seconds and Q_7 takes 16.3 seconds. The runtime for each debugging scenario using Artemis is shown in Fig. 15.

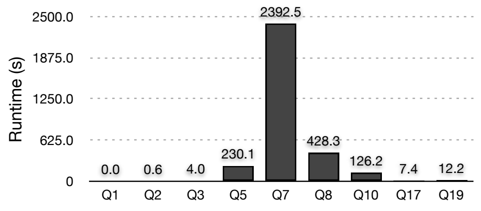


Figure 15: Runtime to compute all explanations using Artemis

Not surprisingly, the runtime of Artemis is significantly higher than the runtime of Missing-Answers. Artemis' constraint-solving phase dominates the total runtime and, thus, scenarios that require a large number of constraint solved are significantly impacted (e.g., Q_5, Q_7, Q_8, and Q_9). Q_7, for instance, requires 23,178 calls to the constraint solver (see Fig. 5). In fact, the main culprit for the large runtimes is the way we implemented the interface between the Artemis runtime and MINION, the constraint-solver. Our current implementation executes MINION as a separate process every time a constraint must be solved. On average, MINION uses 14ms to solve our constraints and, thus, Q_7 uses 325 seconds to resolve all explanations. The remaining 2000s are wasted in system calls.

Artemis is designed as an interactive tool and our emphasis is to quickly produce a small number of initial and correct answers for the user to inspect (We discussed the time needed to return the first explanation in Sec. 6, Exp. 2.). Moreover, using the constraint solver guarantees that each of the returned explanations is correct. This is rather important in cases like Q_7 where without the constraint-solver a large number of incorrect explanations are generate.

Experiment A3: Runtimes for each phase of the algorithm. Fig. 16 shows the fraction of the execution time used by each of the four main steps of Artemis' algorithm to compute all explanations. The "generic witness" part corresponds to the generation of the generic witness, "conditionalize" to Steps 2 and 3 in which the sources and views are converted to c-tables and c-views, "match" to the beginning of Step 4 where tuples in the c-view are matched against missing tuples, and "solve" to the verification of each explanation using the constraint solver. The actual execution times are in Fig 15.

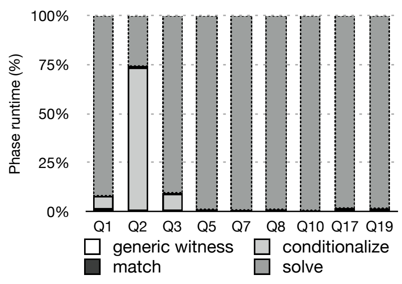


Figure 16: Phase-wise runtime computing all explanations

We clearly see that when computing all explanations, the dominant phase is the solving phase. The exception is Q_2, where conditionalization dominates the total runtime. The reason for this behavior is that Q_2 only performs two calls to the constraint solver. For queries Q_1 and Q_3, the fraction of the conditionalization phase is above 5% because the number of potential explanations to verify is still low, whereas all other queries have to process more than a hundred potential explanations in the solving phase.

Experiment A4: Considering side-effects. All previous experiments did not consider side-effects. I.e., none of the constraints included a C_{im} or C_m component (see Sec. 4.5).

In this experiment, we ran the six debugging scenarios for PhotoShare again (Fig. 12), this time adding restrictions on the number of side-effects allowed.

Fig. 17 shows the total runtime and Fig. 18 shows the number of explanations for each debugging scenario. In the figures, we divided each scenario into three categories: (1) *Standard* – no side-effects are considered, which corresponds to our previous experiments. (2) *No side-effects* – side-effects are not acceptable for the given query and we set $Q_{im} = Q$. (3) *Minimal number of side-effects* – we set $Q_m = Q$.

From Fig. 17, we observe that the difference in runtime when considering side-effects is negligible when the number of potential side-effects (i.e., the size of C_m or C_{im}) is small. For instance, in

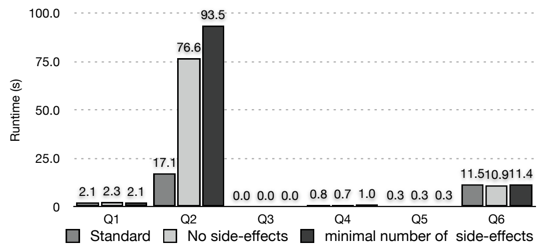


Figure 17: Runtime when including non-empty Q_{im} or Q_m

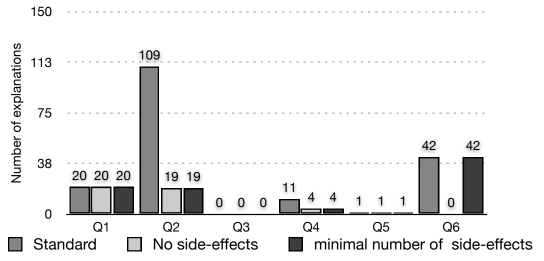


Figure 18: Number of explanations with non-empty Q_{im} or Q_m

scenarios Q_1, Q_4, and Q_6 the number of potential side-effects is 34, 34, and 128, respectively. However, the difference in runtime becomes significant as the number of potential side-effects increases. For instance, scenario Q_2 produces 573 potential side-effects. The runtime for minimizing the number of side-effects is larger than the runtime for avoiding side-effects. This is expected since when no side-effects are allowed, the constraint solver stops processing after the first one is found. As a final remark, Artemis recognizes that scenario Q_3 is unsatisfiable early in the process and finishes it quickly.

Focusing on Fig. 18, we observe that for scenarios Q_1, Q_2, Q_4, and Q_5, the minimum number of side-effects is zero, hence, we obtain the same number of explanations when avoiding side-effects than when minimizing side-effects. For scenarios Q_2 and Q_4, however, some explanations are pruned compared to the case where no side-effects are considered. In scenario Q_6, we observe that the number of explanations with minimal side-effects is equal to the number of all possible explanations, but no explanation is returned when not allowing side-effects. The reason for this is that in this scenario, all explanations generate a side-effect.

E. Proofs Sketches for Propositions

PROOF SKETCH PROP. 1. We first notice that the canonical instance $(R_1(X_1),\ldots,R_k(X_k))[\mathbf{x}/t]$ of any sub-query in Q, where Q is a union of conjunctive queries, is an explanation pattern for t. Indeed, the valuation that maps each variable $x_i \in \mathbf{x}$ to the corresponding value of t is a valuation that will produce the missing tuple t.

Now, take any pattern P that yields t. This means that there is at least one valuation ρ from the body of Q to P such that $\rho(\mathbf{x}) = t$. Assume that the valuation has been obtained for the i-th sub-query in Q, $1 \leq i \leq m$. This means that ρ has a homomorphism from $\left(R_{i_1}(X_{i_1}),\ldots,R_{i_k}(X_{k_i})\right)[\mathbf{x}/t]$ to P. That yields to the conclusion that every possible pattern has a homomorphism from a pattern in the generic witness that consists of the canonical set instance. □

PROOF SKETCH PROP. 2. The proof has two steps. First, we prove that $Core(P_i) = CoreWitness(P_i)$ for SPJU queries and missing tuples with constant values only. We then show that W_{min} is the unique minimum generic witness (up to isomorphism) for W.

Part 1. For select-project-join-union (SPJU) queries, it has been shown that queries are preserved under homomorphism [2]. That is, whenever a tuple $t \in Q(A)$ and a homomorphism $h : A \rightarrow B$ exists, then $h(t) \in Q(B)$. As the following example shows, this property only applies for tuples in a result of a query where all values are constants, whereas we reason about tuples that are not in the result and where t can have both constants and labeled nulls as values.

EXAMPLE 12. *Assume we have a relation R(A,B) with tuple $\langle a,1\rangle$. Further assume we have a query $Q(B_1,B_2)$:- $R(A,B_1),R(A,B_2)$. Clearly, $Q(R) = \langle 1,1\rangle$. Now, assume $t = \langle N_1,N_2\rangle$, which corresponds to t being any possible tuple returned by $Q(R)$. An explanation pattern that can account for any tuple satisfying the constraint defined by t is $P_1 = \langle N_1,N_2\rangle, \langle N_1,N_3\rangle$. It is easy to verify that the pattern $P_2 = \langle N_1,N_2\rangle$ is homomorphically equivalent to P_1, and P_2 is the core of both P_1 and P_2. However, P_2 only accounts for output tuples $\langle N_1,N_1\rangle$ in the result of Q. The equality constraint of both attributes is not part of t, so the core is too specific and only accounts for a subset of tuples that can be witnessed.*

Assuming that t uses constant values only and Q is an SPJU query, it is true that $h(t) = t$ and in this case, if $t \in Q(P)$ and $\exists h : P \rightarrow Core(P)$, then $t \in Q(Core(P))$. Thus, $Core(P)$ is still a valid pattern to generate t and $CoreWitness(P) = Core(P)$.

Part 2. The patterns part of the universal witness can be divided into clusters of patterns, where a cluster is formed by homomorphically equivalent patterns and no two distinct clusters contain patterns that are homomorphically equivalent to each other. Then, for each of these clusters, the core of each pattern in a cluster is the same (up to isomorphism) and the core is still a a valid pattern (result of Part 1). Since two distinct clusters do not share any patterns between which mutual homomorphism exists, their cores are not isomorphic and therefore, every cluster is represented by a different core. We have one unique core for each distinct cluster, so there is a unique combination of these cores. □

PROOF SKETCH PROP. 3. To generate the minimum universal witness, we defined a three-step procedure. Step one ensures that the minimal universal witness consists of core witnesses only, which is necessary by definition. Homomorphically equivalent witnesses have equal cores, and these duplicates are removed in step (ii). As a consequence, all other witnesses either have no homomorphisms between them or there is homomorphism in only one direction. We need to detect the second case and remove the witnesses to which the homomorphism maps to to obtain the minimum universal witness basis. □

PROOF SKETCH PROP. 4. The completeness of E is guaranteed by Steps 2 and 3 of our algorithm (Sec. 4.4). Those two steps generate all possible explanations that match a pattern in the generic witness. It thus remains to show that the generated set is indeed universal.

Every explanation $e \in E$ is a correct explanation (its condition is verified by the constraint solver in Step 4 described in Sec. 4.5) and will produce any missing t tuple w.r.t. the query Q. This means there is a least one valuation ρ from the body of Q, and hence from the generic witness, to e such that replacing each variable in e with the corresponding value produces t, i.e., $\rho(x) = t$. Assume that the valuation has been obtained for the i-th pattern P of the generic witness. This means that ρ has a homomorphism from an explanation e that matches P to e, and e covers e. That yields the conclusion that every possible explanation is covered by an explanation contained in the explanation set E. □