Characterizing and Computing Causes for Query Answers in Databases from Database Repairs and Repair Programs

Leopoldo Bertossi(✉)

School of Computer Science, Carleton University, Ottawa, Canada
bertossi@scs.carleton.ca

Abstract. A correspondence between database tuples as causes for query answers in databases and tuple-based repairs of inconsistent databases with respect to denial constraints has already been established. In this work, answer-set programs that specify repairs of databases are used as a basis for solving computational and reasoning problems about causes. Here, causes are also introduced at the attribute level by appealing to a both null-based and attribute-based repair semantics. The corresponding repair programs are presented, and they are used as a basis for computation and reasoning about attribute-level causes.

1 Introduction

Causality appears at the foundations of many scientific disciplines. In data and knowledge management, the need to represent and compute *causes* may be related to some form of *uncertainty* about the information at hand. More specifically in data management, we need to understand why certain results, e.g. query answers, are obtained or not. Or why certain natural semantic conditions are not satisfied. These tasks become more prominent and difficult when dealing with large volumes of data. One would expect the database to provide *explanations*, to understand, explore and make sense of the data, or to reconsider queries and integrity constraints (ICs). Causes for data phenomena can be seen as a kind of explanations.

Seminal work on *causality in databases* introduced in [32], and building on work on causality as found in artificial intelligence, appeals to the notions of counterfactuals, interventions and structural models [28]. Actually, [32] introduces the notions of: (a) a database tuple as an *actual cause* for a query result, (b) a *contingency set* for a cause, as a set of tuples that must accompany the cause for it to be such, and (c) the *responsibility* of a cause as a numerical measure of its strength (building on [19]).

L. Bertossi—Member of the "Millenium Institute for Foundational Research on Data", Chile.

F. Ferrarotti and S. Woltran (Eds.): FoIKS 2018, LNCS 10833, pp. 55–76, 2018.
https://doi.org/10.1007/978-3-319-90050-6_4

Most of our research on causality in databases has been motivated by an attempt to understand causality from different angles of data and knowledge management. In [11], precise reductions between causality in databases, database repairs, and consistency-based diagnosis were established; and the relationships were investigated and exploited. In [12], causality in databases was related to view-based database updates and abductive diagnosis. These are all interesting and fruitful connections among several forms of non-monotonic reasoning; each of them reflecting some form of uncertainty about the information at hand. In the case of database repairs [8], it is about the uncertainty due the non-satisfaction of given ICs, which is represented by presence of possibly multiple intended repairs of the inconsistent database.

Database repairs can be specified by means of *answer-set programs* (or *disjunctive logic programs with stable model semantics*) [15,26,27], the so-called *repair-programs*. Cf. [8,18] for details on repair-programs and additional references. In this work we exploit the reduction of database causality to database repairs established in [11], by taking advantage of repair programs for specifying and computing causes, their contingency sets, and their responsibility degrees. We show that the resulting *causality-programs* have the necessary and sufficient expressive power to capture and compute not only causes, which can be done with less expressive programs [32], but especially minimal contingency sets and responsibilities (which provably require higher expressive power). Causality programs can also be used for reasoning about causes.

As a finer-granularity alternative to tuple-based causes, we introduce a particular form of *attribute-based causes*, namely *null-based causes*, capturing the intuition that an attribute value may be the cause for a query to become true in the database. This is done by profiting from an abstract reformulation of the above mentioned relationship between tuple-based causes and tuple-based repairs. More specifically, we appeal to *null-based repairs* that are a particular kind of *attribute-based repairs*, according to which the inconsistencies of a database are solved by minimally replacing attribute values in tuples by NULL, the null-value of SQL databases with its SQL semantics. We also define the corresponding notions of contingency set and responsibility. We introduce repair (answer-set) programs for null-based repairs, so that the newly defined causes can be computed and reasoned about.

Finally, we briefly show how causality-programs can be adapted to give an account of other forms of causality in databases that are connected to other possible repair-semantics for databases.

This paper is structured as follows. Section 2 provides background material on relational databases, database causality, database repairs, and answer-set programming (ASP). Section 3 establishes correspondences between causes and repairs, and introduces in particular, null-based causes and repairs. Section 4 presents repair-programs to be used for tuple-based causality computation and reasoning.[1] Section 5 presents answer-set programs for null-based repairs and null-based causes. Finally, Sect. 6, in more speculative terms, contains a

[1] This section is a revised version of the extended abstract [13].

discussion about research subjects that would naturally extend this work. In order to better convey our main ideas and constructs, we present things by means of representative examples. The general formulation is left for the extended version of this paper.

2 Background

2.1 Relational Databases

A relational schema $\mathcal{R}$ contains a domain, $\mathcal{C}$, of constants and a set, $\mathcal{P}$, of predicates of finite arities. $\mathcal{R}$ gives rise to a language $\mathfrak{L}(\mathcal{R})$ of first-order (FO) predicate logic with built-in equality, $=$. Variables are usually denoted by $x, y, z, ...$, and sequences thereof by $\bar{x}, ...$; and constants with $a, b, c, ...$, and sequences thereof by $\bar{a}, \bar{c}, \ldots$. An *atom* is of the form $P(t_1, \ldots, t_n)$, with n-ary $P \in \mathcal{P}$ and $t_1, \ldots, t_n$ *terms*, i.e. constants, or variables. An atom is *ground*, aka. a *tuple*, if it contains no variables. Tuples are denoted with $\tau, \tau_1, \ldots$. A database instance, D, for $\mathcal{R}$ is a finite set of ground atoms; and it serves as a (Herbrand) interpretation structure for language $\mathfrak{L}(\mathcal{R})$ [30] (cf. also Sect. 2.4).

A *conjunctive query* (CQ) is a FO formula of the form $\mathcal{Q}(\bar{x})\colon \exists \bar{y}\ (P_1(\bar{x}_1) \wedge \cdots \wedge P_m(\bar{x}_m))$, with $P_i \in \mathcal{P}$, and (distinct) free variables $\bar{x} := (\bigcup \bar{x}_i) \smallsetminus \bar{y}$. If $\mathcal{Q}$ has n (free) variables, $\bar{c} \in \mathcal{C}^n$ is an *answer* to $\mathcal{Q}$ from D if $D \models \mathcal{Q}[\bar{c}]$, i.e. $Q[\bar{c}]$ is true in D when the variables in $\bar{x}$ are componentwise replaced by the values in $\bar{c}$. $\mathcal{Q}(D)$ denotes the set of answers to $\mathcal{Q}$ from D. $\mathcal{Q}$ is a *boolean conjunctive query* (BCQ) when $\bar{x}$ is empty; and when it is *true* in D, $\mathcal{Q}(D) := \{\mathit{true}\}$. Otherwise, if it is *false*, $\mathcal{Q}(D) := \emptyset$. A *view* is predicate defined by means of a query, whose contents can be computed, if desired, by computing all the answers to the defining query.

In this work we consider integrity constraints (ICs), i.e. sentences of $\mathfrak{L}(\mathcal{R})$, that are: (a) *denial constraints* (DCs), i.e. of the form $\kappa\colon \neg\exists \bar{x}(P_1(\bar{x}_1) \wedge \cdots \wedge P_m(\bar{x}_m))$ (sometimes denoted $\leftarrow P_1(\bar{x}_1), \ldots, P_m(\bar{x}_m)$), where $P_i \in \mathcal{P}$, and $\bar{x} = \bigcup \bar{x}_i$; and (b) *functional dependencies* (FDs), i.e. of the form $\varphi\colon \neg\exists \bar{x}(P(\bar{v}, \bar{y}_1, z_1) \wedge P(\bar{v}, \bar{y}_2, z_2) \wedge z_1 \neq z_2)$. Here, $\bar{x} = \bar{y}_1 \cup \bar{y}_2 \cup \bar{v} \cup \{z_1, z_2\}$, and $z_1 \neq z_2$ is an abbreviation for $\neg z_1 = z_2$.[2] A *key constraint* (KC) is a conjunction of FDs: $\bigwedge_{j=1}^{k} \neg\exists \bar{x}(P(\bar{v}, \bar{y}_1) \wedge P(\bar{v}, \bar{y}_2) \wedge y_1^j \neq y_2^j)$, with $k = |\bar{y}_1| = |\bar{y}_2|$. A given schema may come with its set of ICs, and its instances are expected to satisfy them. If an instance does not satisfy them, we say it is *inconsistent*. In this work we concentrate on DCs, excluding, for example, *inclusion or tuple-generating dependencies* of the form $\forall \bar{x}(\varphi(\bar{x}) \rightarrow \exists \bar{y}\psi(\bar{x}', \bar{y}))$, with $\bar{x}' \subseteq \bar{x}$. See [1] for more details and background material on relational databases.

2.2 Causality in Databases

A notion of *cause* as an explanation for a query result was introduced in [32], as follows. For a relational instance $D = D^n \cup D^x$, where D^n and D^x denote the

[2] The variables in the atoms do not have to occur in the indicated order, but their positions should be in correspondence in the two atoms.

mutually exclusive sets of *endogenous* and *exogenous* tuples, a tuple $\tau \in D^n$ is called a *counterfactual cause* for a BCQ $\mathcal{Q}$, if $D \models \mathcal{Q}$ and $D \smallsetminus \{\tau\} \not\models \mathcal{Q}$. Now, $\tau \in D^n$ is an *actual cause* for $\mathcal{Q}$ if there exists $\Gamma \subseteq D^n$, called a *contingency set* for τ, such that τ is a counterfactual cause for $\mathcal{Q}$ in $D \smallsetminus \Gamma$. This definition is based on [28].

The notion of *responsibility* reflects the relative degree of causality of a tuple for a query result [32] (based on [19]). The responsibility of an actual cause τ for $\mathcal{Q}$, is $\rho(\tau) := \frac{1}{|\Gamma|+1}$, where $|\Gamma|$ is the size of a smallest contingency set for τ. If τ is not an actual cause, $\rho(\tau) := 0$. Intuitively, tuples with higher responsibility provide stronger explanations.

The partition of the database into endogenous and exogenous tuples is because the latter are somehow unquestioned, e.g. we trust them, or we may have very little control on them, e.g. when obtained from an external, trustable and indisputable data source, etc.; whereas the former are subject to experimentation and questioning, in particular, about their role in query answering or violation of ICs. The partition is application dependent, and we may not even have exogenous tuples, i.e. $D^n = D$. *Actually, in the following we will assume all the tuples in a database instance are endogenous.* (Cf. [11] for the general case, and Sect. 6 for additional discussions.) The notion of cause as defined above can be applied to monotonic queries, i.e. whose sets of answers may only grow when the database grows [11].[3] *In this work we concentrate only on conjunctive queries, possibly with built-in comparisons, such as* $\neq$.

Example 1. Consider the relational database $D = \{R(a_4, a_3), R(a_2, a_1), R(a_3, a_3), S(a_4), S(a_2), S(a_3)\}$, and the query $\mathcal{Q}\colon \exists x \exists y (S(x) \land R(x, y) \land S(y))$. D satisfies the query, i.e. $D \models \mathcal{Q}$.

$S(a_3)$ is a counterfactual cause for $\mathcal{Q}$: if $S(a_3)$ is removed from D, $\mathcal{Q}$ is no longer true. So, it is an actual cause with empty contingency set; and its responsibility is 1. $R(a_4, a_3)$ is an actual cause for $\mathcal{Q}$ with contingency set $\{R(a_3, a_3)\}$: if $R(a_4, a_3)$ is removed from D, $\mathcal{Q}$ is still true, but further removing the contingent tuple $R(a_3, a_3)$ makes $\mathcal{Q}$ false. The responsibility of $R(a_3, a_3)$ is $\frac{1}{2}$. $R(a_3, a_3)$ and $S(a_4)$ are actual causes, with responsibility $\frac{1}{2}$. □

2.3 Database Repairs

We introduce the main ideas by means of an example. If only deletions and insertions of tuples are admissible updates, the ICs we consider in this work can be enforced only by deleting tuples from the database, not by inserting tuples (we consider updates of attribute-values in Sect. 3.3). Cf. [8] for a survey on database repairs and consistent query answering in databases.

Example 2. The database $D = \{P(a), P(e), Q(a, b), R(a, c)\}$ is inconsistent with respect to (w.r.t.) the (set of) *denial constraints* (DCs) κ_1: $\neg \exists x \exists y (P(x) \land Q(x, y))$, and κ_2: $\neg \exists x \exists y (P(x) \land R(x, y))$; that is, $D \not\models \{\kappa_1, \kappa_2\}$.

[3] E.g. CQs, unions of CQs (UCQs), Datalog queries are monotonic [11,12].

A *subset-repair*, in short an *S-repair*, of D w.r.t. the set of DCs is a $\subseteq$-maximal subset of D that is consistent, i.e. no proper superset is consistent. The following are S-repairs: $D_1 = \{P(e), Q(a,b), R(a,b)\}$ and $D_2 = \{P(e), P(a)\}$. A *cardinality-repair*, in short a *C-repair*, of D w.r.t. the set of DCs is a maximum-cardinality, consistent subset of D, i.e. no subset of D with larger cardinality is consistent. D_1 is the only C-repair. $\square$

For an instance D and a set Σ of DCs, the sets of S-repairs and C-repairs are denoted with $\mathit{Srep}(D,\Sigma)$ and $\mathit{Crep}(D,\Sigma)$, resp.

2.4 Disjunctive Answer-Set Programs

We consider disjunctive Datalog programs Π with stable model semantics [23], a particular class of answer-set programs (ASPs) [15]. They consist of a set E of ground atoms, called the *extensional database*, and a finite number of rules of the form

$$A_1 \vee \ldots A_n \leftarrow P_1, \ldots, P_m,\ \mathit{not}\ N_1, \ldots,\ \mathit{not}\ N_k, \tag{1}$$

with $0 \leq n, m, k$, and the A_i, P_j, N_s are positive atoms. The terms in these atoms are constants or variables. The variables in the A_i, N_s appear all among those in the P_j.

The constants in program Π form the (finite) Herbrand universe U of the program. The *ground version* of program Π, $\mathit{gr}(\Pi)$, is obtained by instantiating the variables in Π with all possible combinations of values from U. The Herbrand base, HB, of Π consists of all the possible atomic sentences obtained by instantiating the predicates in Π on U. A subset M of HB is a (Herbrand) model of Π if it contains E and satisfies $\mathit{gr}(\Pi)$, that is: For every ground rule $A_1 \vee \ldots A_n \leftarrow P_1, \ldots, P_m,\ \mathit{not}\ N_1, \ldots,\ \mathit{not}\ N_k$ of $\mathit{gr}(\Pi)$, if $\{P_1, \ldots, P_m\} \subseteq M$ and $\{N_1, \ldots, N_k\} \cap M = \emptyset$, then $\{A_1, \ldots, A_n\} \cap M \neq \emptyset$. M is a *minimal model* of Π if it is a model of Π, and no proper subset of M is a model of Π. $\mathit{MM}(\Pi)$ denotes the class of minimal models of Π.

Now, take $S \subseteq \mathit{HB}(\Pi)$, and transform $\mathit{gr}(\Pi)$ into a new, positive program $\mathit{gr}(\Pi) \downarrow S$ (i.e. without *not*), as follows: Delete every ground instantiation of a rule (1) for which $\{N_1, \ldots, N_k\} \cap S \neq \emptyset$. Next, transform each remaining ground instantiation of a rule (1) into $A_1 \vee \ldots A_n \leftarrow P_1, \ldots, P_m$. By definition, S is a *stable model* of Π iff $S \in \mathit{MM}(\mathit{gr}(\Pi) \downarrow S)$. A program Π may have none, one or several stable models; and each stable model is a minimal model (but not necessarily the other way around) [27].

3 Causes and Database Repairs

In this section we concentrate first on *tuple-based causes* as introduced in Sect. 2.2, and establish a reduction to tuple-based database repairs. Next we provide an abstract definition of cause on the basis of an abstract repair-semantics. Finally, we instantiate the abstract semantics to define null-based causes from a particular, but natural and practical notion of attribute-based repair.

3.1 Tuple-Based Causes from Repairs

In [11] it was shown that causes (as represented by database tuples) for queries can be obtained from database repairs. Consider the BCQ $\mathcal{Q}\colon \exists \bar{x}(P_1(\bar{x}_1) \wedge \cdots \wedge P_m(\bar{x}_m))$ that is (possibly unexpectedly) true in D: $D \models \mathcal{Q}$. Actual causes for $\mathcal{Q}$, their contingency sets, and responsibilities can be obtained from database repairs. First, $\neg\mathcal{Q}$ is logically equivalent to the DC:

$$\kappa(\mathcal{Q})\colon \neg\exists \bar{x}(P_1(\bar{x}_1) \wedge \cdots \wedge P_m(\bar{x}_m)). \tag{2}$$

So, if $\mathcal{Q}$ is true in D, D is inconsistent w.r.t. $\kappa(\mathcal{Q})$, giving rise to repairs of D w.r.t. $\kappa(\mathcal{Q})$.

Next, we build differences, containing a tuple τ, between D and S- or C-repairs:

$$\text{(a) } \mathit{Diff}^s(D, \kappa(\mathcal{Q}), \tau) = \{D \smallsetminus D' \mid D' \in \mathit{Srep}(D, \kappa(\mathcal{Q})), \tau \in (D \smallsetminus D')\}, \tag{3}$$

$$\text{(b) } \mathit{Diff}^c(D, \kappa(\mathcal{Q}), \tau) = \{D \smallsetminus D' \mid D' \in \mathit{Crep}(D, \kappa(\mathcal{Q})), \tau \in (D \smallsetminus D')\}. \tag{4}$$

Proposition 1 [11]**.** For an instance D, a BCQ $\mathcal{Q}$, and its associated DC $\kappa(\mathcal{Q})$, it holds:

(a) $\tau \in D$ is an actual cause for $\mathcal{Q}$ iff $\mathit{Diff}^s(D, \kappa(\mathcal{Q}), \tau) \neq \emptyset$.
(b) For each S-repair D' with $(D \smallsetminus D') \in \mathit{Diff}^s(D, \kappa(\mathcal{Q}), \tau)$, $(D \smallsetminus (D' \cup \{\tau\}))$ is a subset-minimal contingency set for τ.
(c) If $\mathit{Diff}^s(D\kappa(\mathcal{Q}), \tau) = \emptyset$, then $\rho(\tau) = 0$. Otherwise, $\rho(\tau) = \frac{1}{|s|}$, where $s \in \mathit{Diff}^s(D, \kappa(\mathcal{Q}), \tau)$ and there is no $s' \in \mathit{Diff}^s(D, \kappa(\mathcal{Q}), \tau)$ with $|s'| < |s|$.
(d) $\tau \in D$ is a most responsible actual cause for $\mathcal{Q}$ iff $\mathit{Diff}^c(D, \kappa(\mathcal{Q}), \tau) \neq \emptyset$. □

Example 3 (Example 1 cont.). With the same instance D and query $\mathcal{Q}$, we consider the DC $\kappa(\mathcal{Q})$: $\neg\exists x \exists y(S(x) \wedge R(x, y) \wedge S(y))$, which is not satisfied by D. Here, $\mathit{Srep}(D, \kappa(\mathcal{Q})) = \{D_1, D_2, D_3\}$ and $\mathit{Crep}(D, \kappa(\mathcal{Q})) = \{D_1\}$, with $D_1 = \{R(a_4, a_3), R(a_2, a_1), R(a_3, a_3), S(a_4), S(a_2)\}$, $D_2 = \{R(a_2, a_1), S(a_4), S(a_2), S(a_3)\}$, $D_3 = \{R(a_4, a_3), R(a_2, a_1), S(a_2), S(a_3)\}$.

For tuple $R(a_4, a_3)$, $\mathit{Diff}^s(D, \kappa(\mathcal{Q}), R(a_4, a_3)) = \{D \smallsetminus D_2\} = \{\{R(a_4, a_3), R(a_3, a_3)\}\}$. So, $R(a_4, a_3)$ is an actual cause, with responsibility $\frac{1}{2}$. Similarly, $R(a_3, a_3)$ is an actual cause, with responsibility $\frac{1}{2}$. For tuple $S(a_3)$, $\mathit{Diff}^c(D, \kappa(\mathcal{Q}), S(a_3)) = \{D \smallsetminus D_1\} = \{S(a_3)\}$. So, $S(a_3)$ is an actual cause, with responsibility 1, i.e. a most responsible cause. □

It is also possible, the other way around, to characterize repairs in terms of causes and their contingency sets [11]. Actually this connection can be used to obtain complexity results for causality problems from repair-related computational problems [11]. Most computational problems related to repairs, especially C-repairs, which are related to most responsible causes, are provably hard. This is reflected in a high complexity for responsibility [11] (cf. Sect. 6 for some more details).

3.2 Abstract Causes from Abstract Repairs

We can extrapolate and abstract out from the characterization of causes of Sect. 3.1 by starting from an abstract *repair-semantics*, $\mathit{Rep}^{\mathsf{S}}(D, \kappa(Q))$, which identifies a class of intended repairs of instance D w.r.t. the DC $\kappa(Q)$. By definition, $\mathit{Rep}^{\mathsf{S}}(D, \kappa(Q))$ contains instances of D's schema that satisfy $\kappa(Q)$. It is commonly the case that those instances depart from D in some pre-specified minimal way, and, in the case of DCs, the repairs in $\mathit{Rep}^{\mathsf{S}}(D, \kappa(Q))$ are all sub-instances of D [8] (In Sect. 3.3, we will depart from this latter assumption.).

More concretely, given a possibly inconsistent instance D, a general class of repair semantics can be characterized through an abstract partial-order relation, $\preceq_D$,[4] on instances of D's schema that is parameterized by D.[5] If we want to emphasize this dependence on the *priority relation* $\preceq_D$, we define the corresponding class of repairs of D w.r.t. a set on ICs $\varSigma$ as:

$$\mathit{Rep}^{\mathsf{S}^{\preceq}}(D, \varSigma) := \{D' \mid D' \models \varSigma, \text{ and } D' \text{ is } \preceq_D\text{-minimal}\}. \tag{5}$$

This definition is general enough to capture different classes of repairs and in relation to different kinds of ICs, e.g. those that delete old tuples and introduce new tuples to satisfy inclusion dependencies, and also repairs that change attribute values. In particular, it is easy to verify that the classes of S- and C-repairs for DCs of Sect. 2.3 are particular cases of this definition.

Returning to a general class of repairs $\mathit{Rep}^{\mathsf{S}}(D, \kappa(Q))$, assuming that repairs are sub-instances of D, and inspired by (3), we introduce:

$$\mathit{Diff}^{\mathsf{S}}(D, \kappa(\mathcal{Q}), \tau) := \{D \smallsetminus D' \mid D' \in \mathit{Rep}^{\mathsf{S}}(D, \kappa(Q)),\ \tau \in (D \smallsetminus D')\}. \tag{6}$$

Definition 1. For an instance D, a BCQ $\mathcal{Q}$, and a class of repairs $\mathit{Rep}^{\mathsf{S}}(D, \kappa(Q))$:

(a) $\tau \in D$ is an *actual* S-*cause* for $\mathcal{Q}$ iff $\mathit{Diff}^{\mathsf{S}}(D, \kappa(\mathcal{Q}), \tau) \neq \emptyset$.
(b) For each $D' \in \mathit{Rep}^{\mathsf{S}}(D, \kappa(Q))$ with $(D \smallsetminus D') \in \mathit{Diff}^{s}(D, \kappa(\mathcal{Q}), \tau)$, $(D \smallsetminus (D' \cup \{\tau\}))$ is an S-contingency set for τ.
(c) The S-responsibility of an actual S-cause is as in Sect. 2.2, but considering only the cardinalities of S-contingency sets $\varGamma$. $\square$

It should be clear that actual causes as defined in Sect. 3.1 are obtained from this definition by using S-repairs. Furthermore, it is also easy to see that each actual S-cause accompanied by one of its S-contingency sets falsifies query $\mathcal{Q}$ in D.

[4] That is, satisfying reflexivity, transitivity and anti-symmetry, namely $D_1 \preceq_D D_2$ and $D_2 \preceq_D D_1 \Rightarrow D_1 = D_2$.

[5] These general *prioritized repairs* based on this kind of priority relations were introduced in [34], where also different priority relations and the corresponding repairs were investigated.

This abstract definition can be instantiated with different repair-semantics, which leads to different notions of cause. In the following subsection we will do this by appealing to attribute-based repairs that change attribute values in tuples by *null*, a null value that is assumed to be a special constant in $\mathcal{C}$, the set of constants for the database schema. This will allow us, in particular, to define causes at the attribute level (as opposed to tuple level) in a very natural manner.[6]

3.3 Attribute-Based Causes

Database repairs that are based on changes of attribute values in tuples have been considered in [7,8,10], and implicitly in [9] to hide sensitive information in a database D via minimal virtual modifications of D. In the rest of this section we make explicit this latter approach and exploit it to define and investigate attribute-based causality (cf. also [11]). First we provide a motivating example.

Example 4. Consider the database instance $D = \{S(a_2), S(a_3), R(a_3, a_1), R(a_3, a_4), R(a_3, a_5)\}$, and the query $\mathcal{Q}$: $\exists x \exists y (S(x) \wedge R(x, y))$. D satisfies $\mathcal{Q}$, i.e. $D \models \mathcal{Q}$.

The three R-tuples in D are actual causes, but clearly the value a_3 for the first attribute of R is what matters in them, because it enables the join, e.g. $D \models S(a_3) \wedge R(a_3, a_1)$. This is only indirectly captured through the occurrence of different values accompanying a_3 in the second attribute of R-tuples as causes for $\mathcal{Q}$.

Now consider the database instance $D_1 = \{S(a_2), S(a_3), R(\mathit{null}, a_1), R(\mathit{null}, a_4), R(\mathit{null}, a_5)\}$, where *null* stands for the null value as used in SQL databases, which cannot be used to satisfy a join. Now, $D' \not\models \mathcal{Q}$. The same occurs with the instances $D_2 = \{S(a_2), S(\mathit{null}), R(a_3, a_1), R(a_3, a_4), R(a_3, a_5)\}$, and $D_3 = \{S(a_2), S(\mathit{null}), R(\mathit{null}, a_1), R(\mathit{null}, a_4), R(\mathit{null}, a_5)\}$, among others that are obtained from D only through changes of attribute values by *null*. □

In the following we assume the special constant *null* may appear in database instances and can be used to verify queries and constraints. We assume that all atoms with built-in comparisons, say $\mathit{null}\, \theta\, \mathit{null}$, and $\mathit{null}\, \theta\, c$, with c a non-null constant, are all false for $\theta \in \{=, \neq, <, >, \ldots\}$. In particular, since a join, say $R(\ldots, x) \wedge S(x, \ldots)$, can be written as $R(\ldots, x) \wedge S(x', \ldots) \wedge x = x'$, it can never be satisfied through *null*. This assumption is compatible with the use of NULL in SQL databases (cf. [10, Sect. 4] for a detailed discussion, also [9, Sect. 2]).

Consider an instance $D = \{\ldots, R(c_1, \ldots, c_n), \ldots\}$ that may be inconsistent with respect to a set of DCs. The allowed repair updates are changes of attribute values by *null*, which is a natural choice, because this is a deterministic solution that appeals to *the* generic data value used in SQL databases to reflect

[6] Cf. also [10, Sects. 4, 5] for an alternative repair-semantics based on both null- and tuple-based repairs w.r.t. general sets of ICs and their repair programs. They could also be used to define a corresponding notion of cause.

the uncertainty and incompleteness in/of the database that inconsistency produces.[7] In order to keep track of changes, we may introduce numbers as first arguments in tuples, as global, unique tuple identifiers (tids). So, D becomes $D = \{\ldots, R(i; c_1, \ldots, c_n), \ldots\}$, with $i \in \mathbb{N}$. The tid is a value for what we call the 0-th attribute of R. With $id(t)$ we denote the id of the tuple $t \in D$, i.e. $id(R(i; c_1, \ldots, c_n)) = i$.

If D is updated to D' by replacement of (non-tid) attribute values by *null*, and the value of the j-th attribute in R, $j > 0$, is changed to *null*, then the change is captured as the string $R[i;j]$, which identifies that the change was made in the tuple with id i in the j-th *position* (or attribute) of predicate R. These strings are collected forming the set:[8]

$$\Delta^{null}(D, D') := \{R[i;j] \mid R(i; c_1, \ldots, c_j, \ldots, c_n) \in D, c_j \neq null, \text{ becomes } R(i; c_1', \ldots, null, \ldots, c_n') \in D'\}.$$

For example, if $D = \{R(1; a, b), S(2; c, d), S(3; e, f)\}$ is changed into $D' = \{R(1; a, null),\ S(2; null, d),\ S(3; null, null)\}$, then $\Delta^{null}(D, D') = \{R[1;2], S[2;1], S[3;1], S[3;2]\}$.

For database instances with the constant *null*, IC satisfaction is defined by treating *null* as in SQL databases, in particular, joins and comparisons in them cannot be satisfied through *null* (cf. [10, Sect. 4] for a precise formal treatment). This is particularly useful to restore consistency w.r.t. DCs, which involve combinations of (unwanted) joins.

Example 5 (Example 1 cont.). Still with instance $D = \{S(a_2), S(a_3), R(a_3, a_1), R(a_3, a_4), R(a_3, a_5)\}$, consider the DC (the negation of $\mathcal{Q}$) $\kappa: \neg\exists x \exists y (S(x) \wedge R(x, z))$. Since $D \not\models \kappa$, D is inconsistent.

The updated instance $D_1 = \{S(a_2), S(null), R(a_3, a_1), R(a_3, a_4), R(a_3, a_5)\}$ (among others updated with *null*) is consistent: $D_1 \models \kappa$. □

Definition 2. A *null-based repair* of D with respect to a set of DCs Σ is a consistent instance D', such that $\Delta^{null}(D, D')$ is minimal under set inclusion.[9] $Rep^{null}(D, \Sigma)$ denotes the class of null-based repairs of D with respect to Σ.[10] A *cardinality-null-based repair* D' minimizes $|\Delta^{null}(D, D')|$. □

[7] Repairs based on updates of attribute values using other constants of the domain have been considered in [35]. We think the developments in this section could be applied to them.

[8] The condition $c_i \neq null$ in its definition is needed in case the initially given instance already contain nulls.

[9] An alternative, but equivalent formulation can be found in [9].

[10] Our setting allows for a uniform treatment of general and combined DCs, including those with (in)equality and other built-ins, FDs, and KCs. However, for the latter and in SQL databases, it is common that `NULL` is disallowed as a value for a key-attribute, among other issues. This prohibition, that we will ignore in this work, can be accommodated in our definition. For a detailed treatment of repairs w.r.t. sets of ICs that include FDs, see [10, Sects. 4, 5].

We can see that the null-based repairs are the minimal elements of the partial order between instances defined by: $D_1 \leq_D^{null} D_2$ iff $\Delta^{null}(D, D_1) \subseteq \Delta^{null}(D, D_2)$.

Example 6. Consider $D = \big\{R(1; a_2, a_1), R(2; a_3, a_3), R(3; a_4, a_3), S(4; a_2), S(5; a_3), S(6; a_4)\big\}$ that is inconsistent w.r.t. the DC

$$\kappa\colon \neg\exists xy(S(x) \wedge R(x, y) \wedge S(y)).$$

Here, the class of null-based repairs, $Rep^{null}(D, \kappa)$, consists of:

$$\begin{aligned}
D_1 &= \{R(1; a_2, a_1), R(2; a_3, a_3), R(3; a_4, a_3), S(4; a_2), S(5; null), S(6; a_4)\},\\
D_2 &= \{R(1; a_2, a_1), R(2; null, a_3), R(3; a_4, null), S(4; a_2), S(5; a_3), S(6; a_4)\},\\
D_3 &= \{R(1; a_2, a_1), R(2; null, a_3), R(3; a_4, a_3), S(4; a_2), S(5; a_3), S(6; null)\},\\
D_4 &= \{R(1; a_2, a_1), R(2; a_3, null), R(3; a_4, null), S(4; a_2), S(5; a_3), S(6; a_4)\},\\
D_5 &= \{R(1; a_2, a_1), R(2; a_3, null), R(3; null, a_3), S(4; a_2), S(5; a_3), S(6; a_4)\},\\
D_6 &= \{R(1; a_2, a_1), R(2; a_3, null), R(3; a_4, a_3), S(4; a_2), S(5; a_3), S(6; null)\}.
\end{aligned}$$

Here, $\Delta^{null}(D, D_2) = \{R[2; 1], R[3; 2]\}$, $\Delta^{null}(D, D_3) = \{R[2; 1],\ S[6; 1]\}$ and $\Delta^{null}(D, D_1) = \{S[5; 1]\}$. The latter is a cardinality-null-based repair. □

According to the motivation provided at the beginning of this section, we can now define causes appealing to the generic construction in (6), and using in it the class of null-based repairs of D. Since repair actions in this case are attribute-value changes, causes can be defined at both the tuple and attribute levels. The same applies to the definition of responsibility. First, inspired by (6), for a tuple $\tau\colon R(i; c_1, \ldots, c_n) \in D$, we introduce:[11]

$$\begin{aligned}
Diff^{null}(D, \kappa(\mathcal{Q}), R[i; c_j]) := \{\Delta^{null}(D, D') \mid D' \in Rep^{null}(D, \kappa(\mathcal{Q})),& \quad (7)\\
R[i; j] \in \Delta^{null}(D, D')\}.&
\end{aligned}$$

Definition 3. For D an instance and $\mathcal{Q}$ a BCQ, and $\tau \in D$ be a tuple of the form $R(i; c_1, \ldots, c_n)$.

(a) $R[i; c_j]$ is a *null-attribute-based (actual) cause* for $\mathcal{Q}$ iff $Diff^{null}(D, \kappa(\mathcal{Q}, R[i; c_j]) \neq \emptyset$, i.e. the value c_j in τ is a cause if it is changed into a null in some repair.
(b) τ is a *null-tuple-based (actual) cause* for $\mathcal{Q}$ if some $R[i; c_j]$ is a *null-attribute-based cause* for $\mathcal{Q}$, i.e. the whole tuple τ is a cause if at least one of its attribute values is changed into a null in some repair.
(c) The responsibility, $\rho^{a\text{-}null}(R[i; c_j])$, of a *null-attribute-based cause* $R[i; c_j]$ for $\mathcal{Q}$, is the inverse of $min\{|\Delta^{null}(D, D')| \ :\ R[i; j] \in \Delta^{null}(D, D'),$ and $D' \in Rep^{null}(D, \kappa(\mathcal{Q}))\}$. Otherwise, it is 0.

[11] This is not a particular case of (6), because it does not contain full tuples.

(d) The responsibility, $\rho^{t\text{-}null}(\tau)$, of a null-tuple-based cause τ for $\mathcal{Q}$, is the inverse of $min\{|\varDelta^{null}(D, D')| \ : \ R[i;j] \in \varDelta^{null}(D, D')$, for some j, and $D' \in Rep^{null}(D, \kappa(\mathcal{Q}))\}$. Otherwise, it is 0. $\square$

In cases (c) and (d) we minimize over the number of changes in a repair. However, in case (d), of a tuple-cause, any change made in one of its attributes is considered in the minimization. For this reason, the minimum may be smaller than the one for a fixed attribute value change; and so the responsibility at the tuple level may be greater than that at the attribute level. More precisely, if $\tau = R(i; c_1, \ldots, c_n) \in D$, and $R[i; c_j]$ is a null-attribute-based cause, then: $\rho^{a\text{-}null}(R[i; c_j]) \leq \rho^{t\text{-}null}(\tau)$.

Example 7 (Example 6 cont.). Consider $R(2; a_3, a_3) \in D$. Its projection on its first (non-id) attribute, $R[2; a_3]$, is a null-attribute-based cause since $R[2; 1] \in \varDelta^{null}(D, D_2)$. Also $R[2; 1] \in \varDelta^{null}(D, D_3)$. Since $|\varDelta^{null}(D, D_2)| = |\varDelta^{null}(D, D_3)| = 2$, we obtain $\rho^{a\text{-}null}(R[2; 1]) = \frac{1}{2}$. Clearly $R(2; a_3, a_3)$ is a null-tuple-based cause for $\mathcal{Q}$, with $\rho^{t\text{-}null}(R(2; a_3, a_3)) = \frac{1}{2}$. $\square$

Example 8 (Example 4 cont.). The instance with tids is $D = \{S(1; a_2), S(2; a_3), R(3; a_3, a_1), R(4; a_3, a_4), R(5; a_3, a_5)\}$. The only null-based repairs are D_1 and D_2, with $\varDelta^{null}(D, D_1) = \{R[3; 1], R[4; 1], R[5; 1]\}$ and $\varDelta^{null}(D, D_2) = \{S[2; 1]\}$.

The values $R[3; a_3], R[4; a_3], R[5; a_3], S[2; a_3]$ are all null-attribute-based causes for $\mathcal{Q}$. Notice that $\rho^{a\text{-}null}(R[3; a_3]) = \rho^{a\text{-}null}(R[4; a_3]) = \rho^{a\text{-}null}(R[5; a_3]) = \frac{1}{3}$, while $\rho^{a\text{-}null}(R[3; a_1]) = \rho^{a\text{-}null}(R[4; a_4]) = \rho^{a\text{-}null}(R[5; a_5]) = 0$, that the value (a_3) in the first arguments of the R-tuples has a non-zero responsibility, while the values in the second attribute have responsibility 0. $\square$

Notice that the definition of tuple-level responsibility, i.e. case (d) in Definition 3, does not take into account that a same id, i, may appear several times in a $\varDelta^{null}(D, D')$. In order to do so, we could redefine the size of the latter by taking into account those multiplicities. For example, if we decrease the size of the $\varDelta$ by one with every repetition of the id, the responsibility for a cause may (only) increase, which makes sense.

In Sect. 5 we will provide repair programs for null-based repairs, which can be used as a basis for specifying and computing null-attribute-based causes.

4 Specifying Tuple-Based Causes

Given a database D and a set of ICs, $\varSigma$, it is possible to specify the S-repairs of D w.r.t. a set $\varSigma$ of DCs, introduced in Sect. 2.3, by means of an answer-set program $\varPi(D, \varSigma)$, in the sense that the set, $Mod(\varPi(D, \varSigma))$, of its stable models is in one-to-one correspondence with $Srep(D, \varSigma)$ [5,18] (cf. [8] for more references). In the following, to ease the presentation, we consider a single denial constraint[12]

$$\kappa\colon \neg\exists\bar{x}(P_1(\bar{x}_1) \wedge \cdots \wedge P_m(\bar{x}_m)).$$

[12] It is possible to consider combinations of DCs and FDs, corresponding to UCQs, possibly with $\neq$, [11].

Although not necessary for S-repairs, it is useful on the causality side having global unique tuple identifiers (tids), i.e. every tuple $R(\bar{c})$ in D is represented as $R(t;\bar{c})$ for some integer t that is not used by any other tuple in D. For the repair program we introduce a nickname predicate R' for every predicate $R \in \mathcal{R}$ that has an extra, final attribute to hold an annotation from the set $\{\mathsf{d},\mathsf{s}\}$, for "delete" and "stays", resp. Nickname predicates are used to represent and compute repairs.

The *repair-ASP*, $\Pi(D,\kappa)$, for D and κ contains all the tuples in D as facts (with tids), plus the following rules:

$$P_1'(t_1;\bar{x}_1,\mathsf{d}) \vee \cdots \vee P_m'(t_n;\bar{x}_m,\mathsf{d}) \leftarrow P_1(t_1;\bar{x}_1),\ldots,P_m(t_m;\bar{x}_m).$$
$$P_i'(t_i;\bar{x}_i,\mathsf{s}) \leftarrow P_i(t_i;\bar{x}_i),\ \mathit{not}\ P_i'(t_i;\bar{x}_i,\mathsf{d}),\ i=1,\cdots,m.$$

A stable model M of the program determines a repair D' of D: $D' := \{P(\bar{c})\ |P'(t;\bar{c},\mathsf{s}) \in M\}$, and every repair can be obtained in this way [18]. For an FD, say $\varphi\colon \neg\exists xyz_1z_2vw(R(x,y,z_1,v) \wedge R(x,y,z_2,w) \wedge z_1 \neq z_2)$, which makes the third attribute functionally depend upon the first two, the repair program contains the rules:

$$R'(t_1;x,y,z_1,v,\mathsf{d}) \vee R'(t_2;x,y,z_2,w,\mathsf{d}) \leftarrow R(t_1;x,y,z_1,v), R(t_2;x,y,z_2,w),$$
$$z_1 \neq z_2.$$
$$R'(t;x,y,z,v,\mathsf{s}) \leftarrow R(t;x,y,z,v),\ \mathit{not}\ R'(t;x,y,z,v,\mathsf{d}).$$

For DCs and FDs, the repair program can be made non-disjunctive by moving all the disjuncts but one, in turns, in negated form to the body of the rule [5,18]. For example, the rule $P(a) \vee R(b) \leftarrow \mathit{Body}$, can be written as the two rules $P(a) \leftarrow \mathit{Body}, \mathit{not}R(b)$ and $R(b) \leftarrow \mathit{Body}, \mathit{not}P(a)$. Still the resulting program can be *non-stratified* if there is recursion via negation [27], as in the case of FDs, and DCs with self-joins.

Example 9 (Example 3 cont.). For the DC $\kappa(\mathcal{Q})\colon \neg\exists x\exists y(S(x) \wedge R(x,y) \wedge S(y))$, the repair-ASP contains the facts (with tids) $R(1;a_4,a_3), R(2;a_2,a_1), R(3;a_3,a_3), S(4;a_4), S(5;a_2), S(6;a_3)$, and the rules:

$$S'(t_1;x,\mathsf{d}) \vee R'(t_2;x,y,\mathsf{d}) \vee S'(t_3;y,\mathsf{d}) \leftarrow S(t_1;x), R(t_2;x,y), S(t_3;y). \quad (8)$$
$$S'(t;x,\mathsf{s}) \leftarrow S(t;x),\ \mathit{not}\ S'(t;x,\mathsf{d}). \quad \text{etc.}$$

Repair D_1 is represented by the stable model M_1 containing $R'(1;a_4,a_3,\mathsf{s}), R'(2;a_2,a_1,\mathsf{s}), R'(3;a_3,a_3,\mathsf{s}), S'(4;a_4,\mathsf{s}), S'(5;a_2,\mathsf{s})$, and $S'(6;a_3,\mathsf{d})$. □

Now, in order to specify causes by means of repair-ASPs, we concentrate, according to (3), on the differences between D and its repairs, now represented by $\{P(\bar{c})\ |\ P(t;\bar{c},\mathsf{d}) \in M\}$, the deleted tuples, with M a stable model of the repair-program. They are used to compute actual causes and their $\subseteq$-minimal contingency sets, both expressed in terms of tids.

The actual causes for the query can be represented by their tids, and can be obtained by posing simple queries to the program under the *uncertain or brave* semantics that makes true what is true in *some* model of the repair-ASP.[13] In this case, $\Pi(D, \kappa(\mathcal{Q})) \models_{brave} \mathit{Cause}(t)$, where the *Cause* predicate is defined on top of $\Pi(D, \kappa(\mathcal{Q}))$ by the rules: $\mathit{Cause}(t) \leftarrow R'(t; x, y, \mathsf{d})$ and $\mathit{Cause}(t) \leftarrow S'(t; x, \mathsf{d})$.

For contingency sets for a cause, given the repair-ASP for a DC $\kappa(\mathcal{Q})$, a new binary predicate $\mathit{CauCont}(\cdot, \cdot)$ will contain a tid for cause in its first argument, and a tid for a tuple belonging to its contingency set. Intuitively, $\mathit{CauCont}(t, t')$ says that t is an actual cause, and t' accompanies t as a member of the former's contingency set (as captured by the repair at hand or, equivalently, by the corresponding stable model). More precisely, for each pair of not necessarily different predicates P_i, P_j in $\kappa(\mathcal{Q})$ (they could be the same if it has self-joins or there are several DCs), introduce the rule $\mathit{CauCont}(t, t') \leftarrow P_i'(t; \bar{x}_i, \mathsf{d}), P_j'(t'; \bar{x}_j, \mathsf{d}), t \neq t'$, with the inequality condition only when P_i and P_j are the same predicate (it is superfluous otherwise).

Example 10 (Examples 3 and 9 cont.). The repair-ASP can be extended with the following rules to compute causes with contingency sets:

$$\mathit{CauCont}(t, t') \leftarrow S'(t; x, \mathsf{d}), R'(t'; u, v, \mathsf{d}).$$
$$\mathit{CauCont}(t, t') \leftarrow S'(t; x, \mathsf{d}), S'(t'; u, \mathsf{d}), t \neq t'.$$
$$\mathit{CauCont}(t, t') \leftarrow R'(t; x, y, \mathsf{d}), S'(t'; u, \mathsf{d}).$$
$$\mathit{CauCont}(t, t') \leftarrow R'(t; x, y, \mathsf{d}), R'(t'; u, v, \mathsf{d}), t \neq t'.$$

For the stable model M_2 corresponding to repair D_2, we obtain $\mathit{CauCont}(1, 3)$ and $\mathit{CauCont}(3, 1)$, from the repair difference $D \smallsetminus D_2 = \{R(a_4, a_3), R(a_3, a_3)\}$. $\square$

We can use extensions of ASP with set- and numerical aggregation to build the contingency set associated to a cause, e.g. the DLV system [29] by means of its DLV-Complex extension [17] that supports set membership and union as built-ins. We introduce a binary predicate *preCont* to hold a cause (id) and a possibly non-maximal set of elements from its contingency set, and the following rules:

$$\begin{aligned}
\mathit{preCont}(t, \{t'\}) &\leftarrow \mathit{CauCont}(t, t'). \\
\mathit{preCont}(t, \#\mathit{union}(C, \{t''\})) &\leftarrow \mathit{CauCont}(t, t''), \mathit{preCont}(t, C), \\
&\qquad \mathit{not}\ \#\mathit{member}(t'', C). \\
\mathit{Cont}(t, C) &\leftarrow \mathit{preCont}(t, C),\ \mathit{not}\ \mathit{HoleIn}(t, C). \\
\mathit{HoleIn}(t, C) &\leftarrow \mathit{preCont}(t, C), \mathit{CauCont}(t, t'), \\
&\qquad \mathit{not}\ \#\mathit{member}(t', C).
\end{aligned}$$

The first two rules build the contingency set for an actual cause (within a repair or stable model) by starting from a singleton and adding additional elements

[13] As opposed to the *skeptical or cautious* semantics that sanctions as true what is true in *all* models. Both semantics as supported by the DLV system [29].

from the contingency set. The third rule, that uses the auxiliary predicate *HoleIn* makes sure that a set-maximal contingency set is built from a pre-contingency set to which nothing can be added.

The responsibility for an actual cause τ, with tid t, as associated to a repair D' (with $\tau \notin D'$) associated to a model M of the extended repair-ASP, can be computed by counting the number of t's for which $\mathit{CauCont}(t,t') \in M$. This responsibility will be maximum within a repair (or model): $\rho(t,M) := 1/(1+|d(t,M)|)$, where $d(t,M) := \{\mathit{CauCont}(t,t') \in M\}$. This value can be computed by means of the *count* function, supported by DLV [24], as follows:

$$\mathit{pre\text{-}rho}(t,n) \leftarrow \#\mathit{count}\{t' : \mathit{CauCont}(t,t')\} = n,$$

followed by the rule computing the responsibility:

$$\mathit{rho}(t,m) \leftarrow m * (\mathit{pre\text{-}rho}(t,n) + 1) = 1.$$

Or, equivalently, via $1/|d(M)|$, with $d(M) := \{P(t';\bar{c},\mathsf{d}) \mid P(t';\bar{c},\mathsf{d}) \in M\}$.

Each model M of the program so far will return, for a given tid that is an actual cause, a *maximal-responsibility contingency set within that model*: no proper subset is a contingency set for the given cause. However, its cardinality may not correspond to the (global) *maximum* responsibility for that tuple. Actually, what we need is $\rho(t) := \max\{\rho(t,M) \mid M \text{ is a model}\}$, which would be an off-line computation, i.e. not within the program. Fortunately, this is not needed since each C-repair gives such a global maximum. So, we need to specify and compute only maximum-cardinality repairs, i.e. C-repairs.

C-repairs can be specified by means of repair-ASPs as above [3], but adding *weak-program constraints* [16,29]. In this case, since we want repairs that minimize the number of deleted tuples, for each database predicate P, we introduce the weak-constraint:

$$:\sim \;\; P(t;\bar{x}),\;\; P'(t;\bar{x},\mathsf{d}).$$

In a model M the body can be satisfied, and then the program constraint violated, but the number of violations is kept to a minimum (among the models of the program without the weak-constraints).[14] A repair-ASP with these weak constraints specifies repairs that minimize the number of deleted tuples; and *minimum-cardinality* contingency sets and maximum responsibilities can be computed, as above.

The approach to specification of causes can be straightforwardly extended via repair programs for several DCs to deal with unions of BCQs (UBCQs), which are also monotonic.

Example 11. Consider $D = \{P(a), P(e), Q(a,b), R(a,c)\}$ and the query $\mathcal{Q} := \mathcal{Q}_1 \vee \mathcal{Q}_2$, with $\mathcal{Q}_1\colon\ \exists xy(P(x) \wedge Q(x,y))$ and $\mathcal{Q}_2\colon \exists xy(P(x) \wedge R(x,y))$. It generates

[14] In contrast, *hard* program-constraints, of the form $\leftarrow \mathit{Body}$, eliminate the models where they are violated, i.e. where *Body* is satisfied. Weak constraints as those above are sometimes denoted with $\Leftarrow P(t;\bar{x}), P'(t;\bar{x},\mathsf{d})$.

the set of DCs: $\Sigma = \{\kappa_1, \kappa_2\}$, with $\kappa_1 \!:\leftarrow P(x), Q(x,y)$ and $\kappa_2 \!:\leftarrow P(x), R(x,y)$. Here, $D \models \mathcal{Q}$ and, accordingly, D is inconsistent w.r.t. Σ.

The actual causes for $\mathcal{Q}$ in D are: $P(a), Q(a,b), R(a,c)$, and $P(a)$ is the most responsible cause. $D_1 = \{P(a), P(e)\}$ and $D_2 = \{P(e), Q(a,b), R(a,c)\}$ are the only S-repairs; D_2 is also the only C-repair for D. The repair program for D w.r.t. Σ contains one rule like (8) for each DC in Σ. The rest is as above in this section. □

Remark 1. When dealing with a set of DCs, each repair rule of the form (8) is meant to solve the corresponding, local inconsistency, even if there is interaction between the DCs, i.e. atoms in common, and other inconsistencies are solved at the same time. However, the minimal-model property of stable models makes sure that in the end a minimal set of atoms is deleted to solve all the inconsistencies [18]. □

5 Specifying Attribute-Based Repairs and Causes

Example 12. Consider the instance $D = \{P(1,2), R(2,1)\}$ for schema $\mathcal{R} = \{P(A,B), R(B,C)\}$. With tuple identifiers it takes the form $D = \{P(1;1,2), R(2;2,1)\}$. Consider also the DC:[15]

$$\kappa\colon\ \neg\exists x \exists y \exists z (P(x,y) \wedge R(y,z)), \tag{9}$$

which is violated by D.

Now, consider the following alternative, updated instances D_i, each them obtained by replacing attribute values by *null*:

D_1	$\{P(1;1,\mathit{null}), R(2;2,1)\}$
D_2	$\{P(1;1,2), R(2;\mathit{null},1)\}$
D_3	$\{P(1;1,\mathit{null}), R(2;\mathit{null},1)\}$

The sets of changes can be identified with the set of changed positions, as in Sect. 3.3, e.g. $\Delta^{null}(D,D_1) = \{P[1;2]\}$ and $\Delta^{null}(D,D_2) = \{R[2;2]\}$ (remember that the tuple id goes always in position 0). These D_i are all consistent, but D_1 and D_2 are the only null-based repairs of D; in particular they are $\leq_D^{null}$-minimal: The sets of changes $\Delta^{null}(D,D_1)$ and $\Delta^{null}(D,D_2)$ are incomparable under set inclusion. D_3 is not $\leq_D^{null}$-minimal, because $\Delta^{null}(D,D_3) = \{P[1;2], R[2;2]\} \supsetneqq \Delta^{null}(D,D_2)$. □

As in Sect. 4, null-based repairs can be specified as the stable models of a disjunctive ASP, the so-called *repair program*. We show next these repair programs by means of Example 12.

The repair-programs for null-based repairs are inspired by ASP-programs that are used to specify virtually and minimally updated versions of a database

[15] It would be easy to consider tids in queries and view definitions, but they do not contribute to the final result and will only complicate the notation. So, we skip tuple ids whenever possible.

D that is protected from revealing certain view contents [9]. This is achieved by replacing direct query answering on D by simultaneously querying (under the certain semantics) the virtual versions of D.

When we have more than one DC, notice that, in contrast to the tuple-based semantics, where we can locally solve each inconsistency without considering inconsistencies w.r.t. other DCs (cf. Remark 1), a tuple that is subject to a local attribute-value update (into *null*) to solve one inconsistency, may need further updates to solve other inconsistencies. For example, if we add in Example 12 the DC $\kappa'\colon \neg\exists x \exists y (P(x,y) \wedge R(y,x))$, the updates in repair D_1 have to be further continued, producing: $P(1; \mathit{null}, \mathit{null}), R(2; \mathit{null}, \mathit{null})$. In other words, every locally updated tuple is considered to: "be in transition" or "being updated" only (not necessarily in a definitive manner) until all inconsistencies are solved.

The above remark motivates the annotation constants that repair programs will use now, for null-based repairs. The intended, informal semantics of annotation constants is shown in the following table. (The precise semantics is captured through the program that uses them.)

Annotation	Atom	The tuple $R(\bar{a})$ ...
$\mathbf{u}$	$R(t; \bar{a}, \mathbf{u})$	Tuple result of an update
$\mathbf{fu}$	$R(t; \bar{a}, \mathbf{fu})$	Final update of a tuple
$\mathbf{t}$	$R(t; \bar{a}, \mathbf{t})$	An initial or updated tuple
$\mathbf{s}$	$R(t; \bar{a}, \mathbf{s})$	Definitive, stays in the repair

More precisely, for each database predicate $R \in \mathcal{R}$, we introduce a copy of it with an extra, final attribute (or argument) that contains an annotation constant. So, a tuple of the form $R(t; \bar{c})$ would become an annotated atom of the form $R'(t; \bar{c}, \mathbf{a})$. The annotation constants are used to keep track of virtual updates, i.e. of old and new tuples: An original tuple $R(t; \bar{c})$ may be successively updated, each time replacing an attribute value by *null*, creating tuples of the form $R(t; \bar{c}', \mathbf{u})$. Eventually the tuple will suffer no more updates, at which point it will become of the form $R'(t; \bar{c}'', \mathbf{fu})$. In the transition, to check the satisfaction of the DCs, it will be combined with other tuples, which can be updated versions of other tuples or tuples in the database that have never been updated. Both kinds of tuples are uniformly annotated with $R'(t', \bar{d}, \mathbf{t})$. In this way, several, possibly interacting DCs can be handled. The tuples that eventually form a repaired version of the original database are those of the form $R'(t; \bar{e}, \mathbf{s})$, and are the final versions of the updated original tuples or the original tuples that were never updated.

In $R'(t; \bar{a}, \mathbf{fu})$, annotation $\mathbf{fu}$ means that the atom with tid t has reached its final update (during the program evaluation). In particular, $R(t; \bar{a})$ has already been updated, and $\mathbf{u}$ should appear in the new, updated atom, say $R'(t; \bar{a}', \mathbf{u})$, and this tuple cannot be updated any further (because relevant updateable attribute values have already been replaced by *null* if necessary). For example, consider a tuple $R(t; a, b) \in D$. A new tuple $R(t; a, \mathit{null})$ is obtained by updating b into *null*. Therefore, $R'(t; a, \mathit{null}, \mathbf{u})$ denotes the updated tuple. If this tuple is not updated any further, it will also eventually appear as $R'(t; a, \mathit{null}, \mathbf{fu})$, indicating it is a final update.[16] (Cf. rules 3. in Example 13.)

The repair program uses these annotations to go through different steps, until its stable models are computed. Finally, the atoms needed to build a repair are read off by restricting a model of the program to atoms with the annotation $\mathbf{s}$. The following example illustrates the main ideas and issues.

Example 13 (Example 12 cont.). Consider $D = \{P(1,2), R(2,1)\}$ and the DC: $\kappa\colon \neg\exists x\exists y\exists z(P(x,y) \wedge R(y,z))$. The repair program $\Pi(D, \{\kappa\})$ is as follows: (it uses several auxiliary predicates to make rules *safe*, i.e. with all their variables appearing in positive atoms in their bodies)

1. $P(1; 1, 2).\ R(2; 2, 1).$ (initial database)
2. $P'(t_1; x, \mathit{null}, \mathbf{u}) \vee R'(t_2; \mathit{null}, z, \mathbf{u}) \leftarrow P'(t_1; x, y, \mathbf{t}),\ R'(t_2; y, z, \mathbf{t}), y \neq \mathit{null}.$
3. $P'(t; x, y, \mathbf{fu}) \leftarrow P'(t; x, y, \mathbf{u}), \mathit{not}\ \mathit{aux}_{P.1}(t; x, y),\ \mathit{not}\ \mathit{aux}_{P.2}(t; x, y).$
   $\mathit{aux}_{P.1}(t; x, y) \leftarrow P'(t; \mathit{null}, y, \mathbf{u}), P(t; x, z), x \neq \mathit{null}.$
   $\mathit{aux}_{P.2}(t; x, y) \leftarrow P'(t; x, \mathit{null}, \mathbf{u}), P(t; z, y), y \neq \mathit{null}.$ (idem for R)
4. $P'(t; x, y, \mathbf{t}) \leftarrow P(t; x, y).$
   $P'(t; x, y, \mathbf{t}) \leftarrow P'(t; x, y, \mathbf{u}).$ (idem for R)
5. $P'(t; x, y, \mathbf{s}) \leftarrow P'(t; x, y, \mathbf{fu}).$ (idem for R)
   $P'(t; x, y, \mathbf{s}) \leftarrow P(t; x, y),\ \mathit{not}\ \mathit{aux}_P(t).$
   $\mathit{aux}_P(t) \leftarrow P'(t; u, v, \mathbf{u}).$

In this program tids in rules are handled as variables. Constant *null* in the program is treated as any other constant. This is the reason for the condition $y \neq \mathit{null}$ in the body of 2, to avoid considering the join through *null* a violation of the DC.[17] A quick look at the program shows that the original tids are never destroyed and no new tids are created, which simplifies keeping track of tuples under repair updates. It also worth mentioning that for this particular example, with a single DC, a much simpler program could be used, but we keep the general form that can be applied to multiple, possibly interacting DCs.

Facts in 1. belong to the initial instance D, and become annotated right away with $\mathbf{t}$ by rules 4. The most important rules of the program are those in 2. They enforce one step of the update-based repair-semantics in the presence of *null* and using *null* (yes, already having nulls in the initial database is not a problem). Rules in 2. capture in the body the violation of DC; and in the head, the intended way of restoring consistency, namely making one of the attributes participating in a join take value *null*.

[16] Under null-based repairs no tuples are deleted or inserted, so the original tids stay all in the repairs and none is created.

[17] If instead of (9) we had $\kappa\colon \neg\exists x\exists y\exists z(P(x,y) \wedge R(y,z) \wedge y < 3)$, the new rule body could be $P'(t_1; x, y, \mathbf{t}), R'(t_2; y, z, \mathbf{t}), y < 3$, because $\mathit{null} < 3$ would be evaluated as false.

Rules in 3. collect the final updated versions of the tuples in the database, as those whose values are never replaced by a *null* in another updated version.

Rules in 4. annotate the original atoms and also new versions of updated atoms. They all can be subject to additional updates and have to be checked for DC satisfaction, with rule 2. Rules in 5. collect the tuples that stay in the final state of the updated database, namely the original and never updated tuples plus the final, updated versions of tuples. In this program *null* is treated as any other constant. □

Proposition 2. There is a one-to-one correspondence between the *null*-based repairs of D w.r.t. a set of DCs Σ and the stable models of the repair program $\Pi(D,\Sigma)$. More specifically, a repair D' can be obtained by collecting the **s**-annotated atoms in a stable model M, i.e. $D' = \{P(\bar{c}) \mid P'(t;\bar{c},\mathbf{s}) \in M\}$; and every repair can be obtained in this way.[18] □

Example 14 (Example 13 cont.). The program has two stable models: (the facts in 1. and the *aux*-atoms are omitted)

$$M_1 = \{P'(1;1,2,\mathbf{t}),\ R'(2;2,1,\mathbf{t}), \underline{R'(2;2,1,\mathbf{s})}, P'(1;1,\mathit{null},\mathbf{u}), P'(1;1,\mathit{null},\mathbf{t}),$$
$$P'(1;1,\mathit{null},\mathbf{fu}), \underline{P'(1;1,\mathit{null},\mathbf{s})}\}.$$
$$M_2 = \{P'(1;1,2,\mathbf{t}),\ R'(2;2,1,\mathbf{t}), \underline{P'(1;1,2,\mathbf{s})}, R'(2;\mathit{null},1,\mathbf{u}), R'(2;\mathit{null},1,\mathbf{t}),$$
$$R'(2;\mathit{null},1,\mathbf{fu}), \underline{R'(2;\mathit{null},1,\mathbf{s})}\}.$$

The repairs are built by selecting the underlined atoms: $D_1 = \{P(1,\mathit{null}), R(2,1)\}$ and $D_2 = \{P(1,2), R(\mathit{null},1)\}$. They coincide with those in Example 12. □

Finally, and similarly to the use of repair programs for cause computation in Sect. 4, we can use the new repair programs to compute null-attribute-based causes (we do not consider here null-tuple-based causes, nor the computation of responsibilities, all of which can be done along the lines of Sect. 4). All we need to do is add to the repair program the definition of a cause predicate, through rules of the form:

$$\mathit{Cause}(t;i;v) \leftarrow R'(t;\bar{x},\mathit{null},\bar{z},\mathbf{s}), R(t;\bar{x}',v,\bar{z}'), v \neq \mathit{null},$$

(with v and *null* the body in the same position i), saying that value v in the i-th position in original tuple with tid t is a null-attribute-based cause. The rule collects the original values (with their tids and positions) that have been changed into *null*. To the program in Example 13 we would add the rules (with similar rules for predicate R)

$$\mathit{Cause}(t;1;x) \leftarrow P'(t;\mathit{null},y,\mathbf{s}), P(t;x,y').$$
$$\mathit{Cause}(t;2;y) \leftarrow P'(t;x,\mathit{null},\mathbf{s}), P(t;x',y).$$

[18] The proof of this claim is rather long, and is similar in spirit to the proof that tuple-based database repairs w.r.t. integrity constraints [6,8] can be specified by means of disjunctive logic programs with stable model semantics (cf. [4,14]).

6 Discussion

Complexity. Computing causes for CQs can be done in polynomial time in data [32], which also holds for UBCQs [11]. In [12] it was established that cause computation for Datalog queries falls in the second level of the polynomial hierarchy (PH). As has been established in [11,32], the computational problems associated to contingency sets and responsibility are at the second level of PH, in data complexity.

On the other side, our repairs programs, and so our causality-ASPs, can be transformed into non-disjunctive, unstratified programs [5,18], whose reasoning tasks are also at the second level of PH (in data) [22]. It is worth mentioning that the ASP approach to causality via repairs programs could be extended to deal with queries that are more complex than CQs or UCQs, e.g. Datalog queries and queries that are conjunctions of literals (that were investigated in [33]).

Causality Programs and ICs. The original causality setting in [32] does not consider ICs. An extension of causality under ICs was proposed in [12]. Under it, the ICs have to be satisfied by the databases involved, i.e. the initial one and those obtained by cause and contingency-set deletions. When the query at hand is monotonic,[19] monotonic ICs, i.e. for which growing with the database may only produce more violations (e.g. denial constraints and FDs), are not much of an issue since they stay satisfied under deletions associated to causes. So, the most relevant ICs are non-monotonic, such as inclusion dependencies, e.g. $\forall xy(R(x,y) \rightarrow S(x))$. These ICs can be represented in a causality-program by means of (strong) program constraints. In the running example, we would have, for tuple-based causes, the constraint: $\leftarrow R'(t,x,y,\mathsf{s}), \mathit{not}\ S'(t',x,\mathsf{s})$.[20]

Negative CQs and Inclusion Dependencies. In this work we investigated CQs, and what we did can be extended to UCQs. However, it is possible to consider queries that are conjunctions of literals, i.e. atoms or negations thereof, e.g. $\mathcal{Q}\colon\ \exists x \exists y(P(x,y) \land \neg S(x))$.[21] (Causes for these queries were investigated in [33].) If causes are defined in terms of counterfactual deletions (as opposed to insertions that can also be considered for these queries), then the repair counterpart can be constructed by transforming the query into the unsatisfied *inclusion dependency* (ID): $\forall x \forall y(P(x,y) \rightarrow S(x))$. Repairs w.r.t. this kind of IDs that allow only tuple deletions were considered in [20], and repairs programs for them in [18]. Causes for CQs in the presence of IDs were considered in [12].

Endogenous and Prioritized Causes and Repairs. As indicated in Sect. 3.2, different kinds of causes can be introduced by considering different repair-semantics. Apart from those investigated in this work, we could consider *endogenous repairs*,

[19] I.e. the set of answers may only grow when the instance grows.

[20] Or better, to make it *safe*, by a rule and a constraint: $\mathit{aux}(x) \leftarrow S'(t',x,\mathsf{s})$ and $\leftarrow R'(t,x,y,\mathsf{s}), \mathit{not}\ \mathit{aux}(x)$.

[21] They should be *safe* in the sense that a variable in a negative literals has to appear in some positive literal too.

which are obtained by removing only (pre-specified) endogenous tuples [11]. In this way we could give an account of causes as in Sect. 2.2, but considering the partition of the database between endogenous and exogenous tuples.

Again, considering the abstract setting of Section 3.2, with the generic class of repairs $Rep^{\mathsf{S}^{\preceq}}(D, \Sigma)$, it is possible to consider different kinds of *prioritized repairs* [34], and through them introduce *prioritized actual causes*. Repair programs for the kinds of priority relations $\preceq$ investigated in [34] could be constructed from the ASPs introduced and investigated in [25] for capturing different optimality criteria. The repair programs could be used, as done in this work, to specify and compute the corresponding prioritized actual causes and responsibilities.

Optimization of Causality Programs. Different queries, but of a fixed form, about causality could be posed to causality programs or directly to the underlying repair programs. Query answering could benefit from query-dependent, magic-set-based optimizations of causality and repair programs as reported in [18]. Implementation and experimentation in general are left for future work.

Connections to Belief Revision/Update. As discussed in [2] (cf. also [8]), there are some connections between database repairs and belief updates as found in knowledge representation, most prominently with [21]. In [3], some connections were established between repair programs and *revision programs* [31]. The applicability of the latter in a causality scenario like ours becomes a matter of possible investigation.

Acknowledgements. This research was supported by NSERC Discovery Grant #06148. Part of this work was done while the author was spending a sabbatical at the "Database and Artificial Intelligence" Group of the Technical University of Vienna with support from the "Vienna Center for Logic and Algorithms" and the Wolfgang Pauli Society. The author is extremely grateful for their support and hospitality, and especially to Prof. Georg Gottlob for making the stay possible. Many thanks to the anonymous reviewers for their excellent feedback.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Boston (1995)
2. Arenas, M., Bertossi, L., Chomicki, J.: Consistent query answers in inconsistent databases. In: Proceedings of PODS, pp. 68–79 (1999)
3. Arenas, M., Bertossi, L., Chomicki, J.: Answer sets for consistent query answers. Theor. Pract. Log. Program. **3**(4&5), 393–424 (2003)
4. Barcelo, P.: Applications of annotated predicate calculus and logic programs to querying inconsistent databases. MSc thesis PUC, Chile (2002). http://people.scs.carleton.ca/~bertossi/papers/tesisk.pdf
5. Barceló, P., Bertossi, L., Bravo, L.: Characterizing and computing semantically correct answers from databases with annotated logic and answer sets. In: Bertossi, L., Katona, G.O.H., Schewe, K.-D., Thalheim, B. (eds.) SiD 2001. LNCS, vol. 2582, pp. 7–33. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36596-6_2

6. Bertossi, L.: Consistent query answering in databases. ACM SIGMOD Rec. **35**(2), 68–76 (2006)
7. Bertossi, L., Bravo, L., Franconi, E., Lopatenko, A.: The complexity and approximation of fixing numerical attributes in databases under integrity constraints. Inf. Syst. **33**(4), 407–434 (2008)
8. Bertossi, L.: Database Repairing and Consistent Query Answering. Synthesis Lectures on Data Management. Morgan & Claypool, San Rafael (2011)
9. Bertossi, L., Li, L.: Achieving data privacy through secrecy views and null-based virtual updates. IEEE Trans. Knowl. Data Eng. **25**(5), 987–1000 (2013)
10. Bertossi, L., Bravo, L.: Consistency and trust in peer data exchange systems. Theor. Pract. Log. Program. **17**(2), 148–204 (2017)
11. Bertossi, L., Salimi, B.: From causes for database queries to repairs and model-based diagnosis and back. Theor. Comput. Syst. **61**(1), 191–232 (2017)
12. Bertossi, L., Salimi, B.: Causes for query answers from databases: datalog abduction, view-updates, and integrity constraints. Int. J. Approx. Reason. **90**, 226–252 (2017)
13. Bertossi, L.: The causality/repair connection in databases: causality-programs. In: Moral, S., Pivert, O., Sánchez, D., Marín, N. (eds.) SUM 2017. LNCS (LNAI), vol. 10564, pp. 427–435. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67582-4_33
14. Bravo, L.: Handling inconsistency in databases and data integration systems. Ph.D. thesis, Carleton University, Department of Computer Science (2007). http://people.scs.carleton.ca/~bertossi/papers/Thesis36.pdf
15. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 93–103 (2011)
16. Buccafurri, F., Leone, N., Rullo, P.: Enhancing disjunctive datalog by constraints. IEEE Tran. Knowl. Data Eng. **12**(5), 845–860 (2000)
17. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: An ASP system with functions, lists, and sets. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS (LNAI), vol. 5753, pp. 483–489. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04238-6_46
18. Caniupan-Marileo, M., Bertossi, L.: The consistency extractor system: answer set programs for consistent query answering in databases. Data Know. Eng. **69**(6), 545–572 (2010)
19. Chockler, H., Halpern, J.Y.: Responsibility and blame: a structural-model approach. J. Artif. Intell. Res. **22**, 93–115 (2004)
20. Chomicki, J., Marcinkowski, J.: Minimal-change integrity maintenance using tuple deletions. Inf. Comput. **197**(1–2), 90–121 (2005)
21. Chou, T., Winslett, M.: A model-based belief revision system. J. Autom. Reason. **12**, 157–208 (1994)
22. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Comput. Surv. **33**(3), 374–425 (2001)
23. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Trans. Database Syst. **22**(3), 364–418 (1997)
24. Faber, W., Pfeifer, G., Leone, N., Dell'Armi, T., Ielpa, G.: Design and implementation of aggregate functions in the DLV system. Theor. Pract. Log. Program. **8**(5–6), 545–580 (2008)
25. Gebser, M., Kaminski, R., Schaub, T.: Complex optimization in answer set programming. Theor. Pract. Log. Program. **11**(4–5), 821–839 (2011)

26. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, San Rafael (2012)
27. Gelfond, M., Kahl, Y.: Knowledge Representation and Reasoning, and the Design of Intelligent Agents. Cambridge University Press, Cambridge (2014)
28. Halpern, J., Pearl, J.: Causes and explanations: a structural-model approach: part 1. Br. J. Philos. Sci. **56**, 843–887 (2005)
29. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Log. **7**(3), 499–562 (2006)
30. Lloyd, J.W.: Foundations of Logic Programming. Springer, Heidelberg (1987). https://doi.org/10.1007/978-3-642-83189-8
31. Marek, V., Truszczynski, M.: Revision programming. Theor. Comput. Sci. **190**(2), 241–277 (1998)
32. Meliou, A., Gatterbauer, W., Moore, K.F., Suciu, D.: The complexity of causality and responsibility for query answers and non-answers. Proc. VLDB Endow. **4**(1), 34–45 (2010)
33. Salimi, B., Bertossi, L., Suciu, D., Van den Broeck, G.: Quantifying causal effects on query answering in databases. In: Proceedings of TaPP (2016)
34. Staworko, S., Chomicki, J., Marcinkowski, J.: Prioritized repairing and consistent query answering in relational databases. Ann. Math. Artif. Intell. **64**(2–3), 209–246 (2012)
35. Wijsen, J.: Database repairing using updates. ACM Trans. Database Syst. **30**(3), 722–768 (2005)