On Multiple Semantics for Declarative Database Repairs

Amir Gilad
Tel Aviv University
amirgilad@mail.tau.ac.il

Daniel Deutch
Tel Aviv University
danielde@post.tau.ac.il

Sudeepa Roy
Duke University
sudeepa@cs.duke.edu

ABSTRACT

We study the problem of database repairs through a rule-based framework that we refer to as *Delta Rules*. Delta rules are highly expressive and allow specifying complex, cross-relations repair logic associated with Denial Constraints, Causal Rules, and allowing to capture Database Triggers of interest. We show that there are no one-size-fits-all semantics for repairs in this inclusive setting, and we consequently introduce multiple alternative semantics, presenting the case for using each of them. We then study the relationships between the semantics in terms of their output and the complexity of computation. Our results formally establish the tradeoff between the permissiveness of the semantics and its computational complexity. We demonstrate the usefulness of the framework in capturing multiple data repair scenarios for an academic search database and the TPC-H databases, showing how using different semantics affects the repair in terms of size and runtime, and examining the relationships between the repairs. We also compare our approach with SQL triggers and a state-of-the-art data repair system.

KEYWORDS

Database Constraints, Provenance, Repairs, Triggers

ACM Reference Format:
Amir Gilad, Daniel Deutch, and Sudeepa Roy. 2020. On Multiple Semantics for Declarative Database Repairs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3389721

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389721

1 INTRODUCTION

The problem of data repair has been extensively studied by previous work [5, 6, 10, 11, 19, 44]. Many of these have focused on the desideratum of minimum cardinality, i.e., repairing the database while making the minimum number of changes [5, 19, 34]. In particular, when the repair only involves tuple deletion [10, 33, 34], this desideratum takes center stage since a näive repair could simply delete the entire database in order to repair it. Such repairs are commonly used with classes of constraints such as Denial Constraints (DCs) [10, 11], SQL deletion triggers [22], and causal dependencies [46].

Different scenarios, however, may require different interpretations of the constraints and the manner in which they should be used to achieve a minimum repair. For integrity constraints such as DCs, when there is a set of tuples violating such a DC, any tuple in that set is a '*candidate for deletion*' to repair the database. Moreover, if we allow such constraints to be influenced by deleted tuples, as needed in cascade deletions, the problem becomes more convoluted.

In contrast, for violations of referential integrity constraints under cascade delete semantics, or other complex and user-defined constraints as in SQL triggers and in causal dependencies, there is a specific tuple that is meant to be deleted if a trigger or a rule is satisfied. Nevertheless, if there are several triggers or causal rules, all satisfied at the same time, it remains largely unspecified and varies from system to system in what order they should be fired and when should the database be updated. For instance, by default, MySQL chooses to fire triggers in the order they have been created [40], and PostgreSQL fires triggers in alphabetical order in such scenarios [42]. This may lead to different answers leaving users uncertain about why the tuples have been deleted. These systems offer an option of specifying the order in which the triggers would fire; however, this order does not guarantee a consistent semantics that leads to a minimum repair. Such constraints may also follow several different semantics in the process of cascading deletions. Therefore, the same set of constraints may be assigned different reasonable semantics that lead to different minimum repairs, and each choice of semantics may be suitable for a different setting.

Example 1.1. *Consider the database in Figure 1 based on an academic database [35]. It contains the tables* Grant *(grant foundations),* Author *(paper authors),* AuthGrant *(a relationship of authors and grants given by a foundation),* Pub *(a publication table),* Writes *(a relationship table between* Author *and* Pub*), and* Cite *(a citation table of citing and cited relationships). For each tuple, we also have an identifier on the leftmost column of each table (e.g.,* ag_1 *is the identifier of* AuthGrant(2, 1)*). Consider the following four constraints specifying how to repair the tables (there could be other rules capturing different repair scenarios):*

(1) If a Grant *tuple is deleted and there is an author who won a grant by this foundation, denoted as an* AuthGrant *tuple, then delete the winning author.*

(2) If an Author *tuple is deleted and the corresponding* Writes *and* Pub *tuples exist in the database, delete the corresponding* Writes *tuple (as in cascade delete semantics for foreign keys).*

(3) Under the same condition as above, delete the corresponding Pub *tuple (not standard foreign keys, but suggesting that every author is important for a publication to exist).*

(4) If a publication p from the Pub *table is deleted, and is cited by another publication c, while some authors of these papers still exist in the database, then delete the* Cite *tuple*[1].

Suppose we are analyzing a subset of this database containing only authors affiliated with U.S. schools and only papers written solely by U.S. authors. ERC grants are given only to European institutions and its Grant *tuple was incorrectly added to the U.S. database, so this tuple* g_2 *needs to be deleted. However, this deletion causes violations in the above constraints. To repair the database based on these constraints, we could proceed in various ways: considering the semantics of triggers and causal rules, we can delete tuples* a_2, w_1, p_1, a_3, w_2, p_2 *and* c*, and regain the integrity of the database but at the cost of deleting seven tuples. A different approach is to delete* a_2 *and either* w_1 *or* p_1*, and delete* a_3 *and either* w_2 *or* p_2*, which would only delete four tuples. However, if we consider the semantics of DCs, we could delete any tuple out of the set of tuples that violates the constraint. So, we can just delete the tuples* $\mathsf{ag}_2, \mathsf{ag}_3$*. This would satisfy the first constraint and thus the second, third and fourth constraints will also be satisfied.*

Our Contributions

In this paper, we propose a novel unified constraint specification framework, with multiple alternative semantics that can be suitable for different settings, and thus can result in different 'minimum repairs'. Our framework allows for semantics similar to DCs as well as causal rules, and the subset of SQL triggers that delete tuple(s) after another deletion event, and is geared toward minimum database repair using tuple deletions.

Grant

	gid	name
g_1	1	NSF
g_2	2	ERC

AuthGrant

	aid	gid
ag_1	2	1
ag_2	4	2
ag_3	5	2

Author

	aid	name
a_1	2	Maggie
a_2	4	Marge
a_3	5	Homer

Cite

	citing	cited
c	7	6

Writes

	aid	pid
w_1	4	6
w_2	5	7

Pub

	pid	title
p_1	6	x
p_2	7	y

Figure 1: Academic database instance D

(0) $\Delta_{\mathsf{Grant}}(g, n)$:- Grant(g, n), $n = \text{'}ERC\text{'}$
(1) $\Delta_{\mathsf{Author}}(a, n)$:- Author(a, n), AuthGrant(a, g), $\Delta_{\mathsf{Grant}}(g, gn)$
(2) $\Delta_{\mathsf{Pub}}(p, t)$:- Pub(p, t), Writes(a, p), $\Delta_{\mathsf{Author}}(a, n)$
(3) $\Delta_{\mathsf{Writes}}(a, p)$:- Pub(p, t), Writes(a, p), $\Delta_{\mathsf{Author}}(a, n)$
(4) $\Delta_{cite}(c, p)$:- Cite(c, p), $\Delta_{\mathsf{Pub}}(p, t)$, Writes$(a_1, c)$, Writes$(a_2, p)$

Figure 2: Delta program

Delta rules and stabilizing sets. We begin by defining the concept of delta rules. Delta rules allow for a deletion of a tuple from the database if certain conditions hold. Intuitively, delta rules are constraints specifying conditions that, if satisfied, compromise the integrity of the database. A stabilizing set is a set of tuples whose removal from the database ensures that no delta rules are satisfied.

Example 1.2. *Reconsider Example 1.1 where the constraints are specified verbatim. We can formalize them in our declarative syntax, as shown in Figure 2. Rules (1), (2), (3), and (4) express the constraints in Example 1.1, respectively. For example, rule (3) states that if a* Pub *and a* Writes *tuples exist in the database, and the corresponding* Author *tuple has been deleted (the* $\Delta_{\mathsf{Author}}(a, n)$ *atom), then delete the* Pub *tuple (this is the head of the rule). Rule (0) initializes the deletion process (more details about this in Section 3). In Example 1.1,* $\{\mathsf{g}_2, \mathsf{a}_2, \mathsf{a}_3, \mathsf{w}_1, \mathsf{w}_2, \mathsf{p}_1, \mathsf{p}_2, \mathsf{c}\}$, $\{\mathsf{g}_2, \mathsf{a}_2, \mathsf{a}_3, \mathsf{w}_1, \mathsf{w}_2, \mathsf{p}_1, \mathsf{p}_2\}$, $\{\mathsf{g}_2, \mathsf{a}_2, \mathsf{a}_3, \mathsf{w}_1, \mathsf{w}_2\}$*, and* $\{\mathsf{g}_2, \mathsf{ag}_2, \mathsf{ag}_3\}$ *are all stabilizing sets, as a removal of any of these sets of tuples and an addition of these tuples to the delta relations ensure that no delta rules are satisfied.*

Although we can easily verify that the deletion of any set of tuples in Example 1.2 guarantees that the database is 'stable', it may not be immediately obvious under what scenarios we would obtain these sets as the answer, or whether they correspond to some notion of 'optimal repair'.

[1] An alternative version of this constraint is not conditioned by the existence of the paper authors, however, the condition is added to demonstrate a difference between the semantics in our framework

Semantics of delta rules. To address this, we define four semantics of delta rules and define the minimum repair according to these. A semantics in this context implies a manner in which we interpret the rules, either as integrity constraints for which we define a global minimum solution, or as means of deriving tuples in different ways. *Independent semantics* aims at finding the globally optimum repair such that none of the rules are satisfied on the entire database instance. It is similar to optimal repair in presence of DCs like violations of functional dependencies [10], but delta rules capture more general propagations of conflict resolutions, where deleting one tuple to resolve a conflict may lead to deletion of another tuple. *Step semantics*, is geared towards the semantics of the aforementioned subset of SQL triggers and causal rules, and is a fine-grained semantics. It evaluates one rule at a time (non-deterministically) and updates the database immediately by removing the tuple, which may in turn lead to further tuple deletions. *Stage semantics* also aims to capture triggers and causal rules. However, as opposed to step semantics, it deterministically removes tuples in stages. In particular, it evaluates all delta rules based on the stage of the database in the previous round, and therefore the order of firing the rules does not matter (similar to seminäive evaluation of datalog [4]). Finally, *end semantics* is similar to the standard datalog evaluation, where all possible delta tuples are first derived and the database is updated at the end of the evaluation process. We use end semantics as a baseline for the other semantics.

Example 1.3. *Continuing Example 1.2, the results corresponding to different semantics are* $End(P, D) = \{g_2, a_2, a_3, w_1, w_2, p_1, p_2, c\}$, $Stage(P, D) = \{g_2, a_2, a_3, w_1, w_2, p_1, p_2\}$, $Step(P, D) = \{g_2, a_2, a_3, w_1, w_2\}$, *and* $Ind(P, D) = \{g_2, ag_2, ag_3\}$. *We detail the formal definitions of the semantics in Section 3.*

Relationships between the results of different semantics. We study the relationships of containment and size between the results according to the four semantics. The results are summarized in Figure 3, where the size of the result of independent semantics is always smaller or equal to the sizes of the results of stage and step semantics. We show there are case where the result of step semantics subsumes the result of stage semantics and vice-versa.

Complexity of finding the results We show that finding the result for stage and end semantics is PTIME, while finding the result for step and independent semantics is NP-hard (also shown in Figure 3). For independent semantics, we devise an efficient algorithm using *data provenance*, leveraging a reduction to the min-ones SAT problem [31]. We store the provenance [25] as a Boolean formula and find a satisfying assignment that maps the minimum number of negated variables to True. For step semantics, we also devise

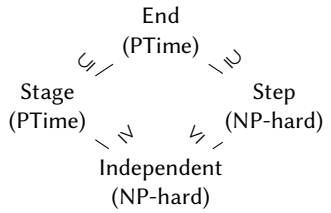


Figure 3: Complexity and relationships among the different semantics by size and containment

an efficient algorithm based on the structure of the provenance graph, traversing it in topological order and choosing tuples for the result set as we go.

Experimental evaluation. We examine the performance of our algorithms for a variety of programs with varying degree of complexity on an academic dataset [35] and the TPC-H dataset [50]. We measure the performance in terms of subsumption relationship between the results computed under different semantics, the size of these results, the execution time of each algorithm to compute the result for every semantics. Finally, for our heuristic algorithms, we break down the execution time in the context of multiple "classes" of programs. We also compare our approach to SQL triggers in PostgreSQL and MySQL, and to `HoloClean` [44].

2 PRELIMINARIES

We start by reviewing basic definitions for databases and non-recursive datalog programs. A relational schema is a tuple $\mathbf{R} = (R_1, \ldots, R_k)$ where R_i is a relation name (or atom). Each relation R_i ($i = 1$ to k) has a set of attributes $\mathbb{A}_i$, and we use $\mathbb{A} = \cup_i \mathbb{A}_i$ to denote the set of all attributes in $\mathbf{R}$. For any attribute $A \in \mathbb{A}$, $\mathtt{dom}(A)$ denotes the domain of A. A database instance D is a finite set of tuples over $\mathbf{R}$, and we will use $R_1, \cdots, R_k$ to denote both the relation names and their content in D where it is clear from the context.

Non-recursive datalog. We will use standard datalog program comprising rules of the form $Q(\mathbf{X}) :- T_{i_1}(\mathbf{Y}_{i_1}), \ldots, T_{i_\ell}(\mathbf{Y}_{i_\ell})$, where $\mathbf{Y_{i_1}}, \cdots, \mathbf{Y_{i_\ell}}$ contain variables or constants, and $\mathbf{X}$ is a subset of the variables in $\cup_{j=1}^{\ell} \mathbf{Y_{i_j}}$. In this rule, Q is called an *intensional* (or derived) relation, and T_i's are either intensional relations or are *extensional* (or base) relations from $\mathbf{R}$. For brevity, we use the notations $body(Q)$ for the set $\{T_{i_1}(\mathbf{Y}_1), \ldots, T_{i_\ell}(\mathbf{Y}_{i_\ell})\}$, and $head(Q)$ for $Q(\mathbf{X})$. A datalog program is simply a set of datalog rules. In this paper, we consider programs $P = \{r_1, \ldots, r_m\}$ such that for some i, j, the relation name of $head(r_i)$ is an element in $body(r_j)$, but P is equivalent to a non-recursive program. These are called bounded programs and are not inherently recursive [4].

Let D be a database and $Q(\mathbf{X}) :- T_{i_1}(\mathbf{Y}_{i_1}), \ldots, T_{i_\ell}(\mathbf{Y}_{i_\ell})$ be a datalog rule, both over the schema $\mathbf{R}$ (i.e., $\forall R_i \in body(Q)$. $R_i \in \mathbf{R}$). An assignment to Q is a function α :

$body(Q) \rightarrow D$ that respects relation names. We require that a variable y_j will not be mapped to multiple distinct values, and a constant y_j will be mapped to itself. We define $\alpha(head(Q))$ as the tuple obtained from $head(Q)$ by replacing each occurrence of a variable x_i by $\alpha(x_i)$.

Given a database D and a datalog program P, we say that it has reached a *fixpoint* if no more tuples can be added to the result set using assignments from D to the rules of P. The fixpoint, denoted by $P(D)$, is then the database obtained by adding to D all tuples derived from the rules of P.

Example 2.1. *Consider the database D in Figure 1 and the program P in Figure 2, and consider for now all Δ relations as standard intensional relations. When the rules are evaluated over the database, after deriving $\Delta_{\mathsf{Grant}}(2, ERC)$ from rule (0), we have two assignments to rule (1): α_1, α_2, where α_1 (α_2) maps the first, second and third atoms to a_2 (a_3), ag_2 (ag_3), and $\Delta_{\mathsf{Grant}}(2, ERC)$ respectively, which generate $\Delta_{\mathsf{Author}}(4, Marge)$ and $\Delta_{\mathsf{Author}}(5, Homer)$. Next we have two assignments to rule (2): α_3, α_4, where α_3 (α_4) maps the first, second and third atoms to p_2 (p_3), w_1 (w_2), and $\Delta_{\mathsf{Author}}(4, Marge)$ ($\Delta_{\mathsf{Author}}(5, Homer)$) respectively. The fixpoint of this evaluation process is the database $P(D) = D \cup \{\Delta_{\mathsf{Grant}}(2, ERC), \Delta_{\mathsf{Author}}(4, Marge), \Delta_{\mathsf{Author}}(5, Homer), \Delta_{\mathsf{Writes}}(4, 6), \Delta_{\mathsf{Writes}}(5, 7), \Delta_{\mathsf{Pub}}(6, x), \Delta_{\mathsf{Pub}}(7, y), \Delta_{\mathsf{Cite}}(7, 6)\}$. This evaluation corresponds to* end semantics *as discussed later.*

3 FRAMEWORK FOR DELTA RULES

We now formulate the model used in the rest of the paper.

3.1 Delta Relations, Rules, and Program

Delta Relations. Given a schema $\mathbf{R} = (R_1, \ldots, R_k)$ where R_i has attributes $\mathbb{A}_i$, the delta relations $\mathbf{\Delta} = (\Delta_1, \ldots, \Delta_k)$ will be used to capture tuples to be deleted from $R_1, \ldots .R_k$ respectively. Therefore, each relation Δ_i has the same set of attributes $\mathbb{A}_i$ (the 'full' notation for Δ_i is Δ_{R_i}, but we abbreviate it).

Delta rules and program. A delta program is a datalog program where every intensional relation is of the form Δ_i for some i.

Definition 3.1. *Given a schema $\mathbf{R} = (R_1, \ldots, R_k)$ and the corresponding delta relations $\mathbf{\Delta} = (\Delta_1, \ldots, \Delta_k)$, a delta rule is a datalog rule of the form $\Delta_i(\mathbf{X}) :- R_i(\mathbf{X}), Q_1(\mathbf{Y}_1), \ldots, Q_l(\mathbf{Y}_l)$ where $Q_i \in \mathbf{R} \cup \mathbf{\Delta}$.*

Intuitively, the condition $Q_i \in \mathbf{R} \cup \mathbf{\Delta}$ means that delta rules can have cascaded deletions when some of the other tuples are removed. Note that the same vector $\mathbf{X}$ that appears in the head Δ_i, also appears in the body in the atom with relation R_i. This is because we need the atom $R_i(X)$ in the body of the rule so that we only delete existing facts. Also, Y_i can intersect with X and any other Y_j. We will refer to a set of delta rules as a *delta program.*

Example 3.2. *Consider rule (2) in Figure 2. This rule is meant to delete any* Pub *tuple after its* Author *tuple has been deleted, intuitively saying that if an author of a paper was deleted, then her associated papers should be deleted as well. We have $\Delta_{\mathsf{Pub}}(p, t)$ in the head of the rule and in the body we have the atom $\mathsf{Pub}(p, t)$ to make sure the deleted tuple exists in the database and we have a join between the* Pub *atom and the $\Delta_{\mathsf{Author}}(a, n)$ atom using the atom $\mathtt{Writes}(a, p)$.*

Overloading notation, we shall use Δ also as a mapping from any subset of tuples in $\mathbf{R}$ to $\mathbf{\Delta}$ in the instance D. For instance, for two tuples from R_1, R_2 as $S = \{R_1(a), R_2(b)\}$, we will use $\Delta(S)$ to denote $\Delta_1(a)$ and $\Delta_2(b)$ suggesting that these two tuples have been deleted.

3.2 Independent Semantics

This non-operational 'ideal' semantics captures the intuition of a minimum-size repair: the smallest set of tuples that need to be removed so that all the constraints are satisfied. Note that whenever we delete a tuple, we add the corresponding delta tuple. Hence the following definition:

Definition 3.3. *Let D and P respectively be a database instance and a delta program over the schema $\mathbf{R}, \Delta$. The result of independent semantics, denoted $Ind(P, D)$, is the smallest subset of non-delta tuples $S \subseteq D$ such that in $(D \setminus S) \cup \Delta(S)$ there is no satisfying assignment for any rule of P.*

Note that there may be multiple minimum size sets satisfying the criteria, in which case the independent semantics will non-deterministically output one of them. Proposition 3.18 shows that there is always a result for this semantics.

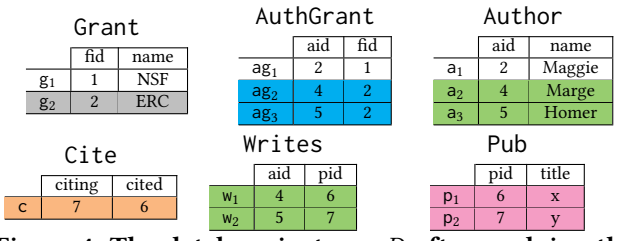


Figure 4: The database instance D after applying the rules in Figure 2 with the different semantics (not showing delta relations). The tuple g_2 is always deleted and added to the delta relation. Tuples of a certain color are deleted from the original relations and added to the delta relations. (1) Independent semantics deletes the gray and cyan tuples. (2) Step semantics deletes the gray and green tuples. (3) Stage semantics deletes the gray, green and pink tuples. (4) End semantics deletes the gray, green, pink, and orange tuples and adds them to the delta relations

Example 3.4. *Consider the database in Figure 1 and the rules shown in Figure 2. The result of independent semantics is* $\{g_2, ag_2, ag_3\}$ *and the final state of the database appears in Figure 4, where the gray and cyan colored tuples are deleted from the original relations and added to the delta relations. Note that in the state depicted in Figure 4, there are no satisfying assignments to any of the rules in Figure 2.*

3.3 Step Semantics

This semantics offers a non-deterministic fine-grain rule activation similar to the fact-at-a-time semantics for datalog in the presence of functional dependencies [2, 3]. We denote the *state* of the database at step t by $D^t = \{R_i^t\}, \Delta^t = \{\Delta_i^t\}, i = 1$ to m, and inductively define step semantics as follows.

Definition 3.5. *Let D and P be a database and a delta program over schema $\mathbf{R} \cup \mathbf{\Delta}$. In step semantics, at step $t = 0$, we have $\Delta_i^t = \emptyset$ and R_i^t = the relation R in D. For each step $t > 0$, make a non-deterministic choice of an assignment $\alpha : body(r) \rightarrow D^t$ to a rule $r \in P$ such that $head(r) = \Delta_i(\mathbf{X})$, $tup = \alpha(head(r))$, and update $\Delta_i^{t+1} \leftarrow \Delta_i^t \cup \{tup\}$, and $R_i^{t+1} \leftarrow R_i^t \setminus \Delta_i^{t+1}$. For $j \neq i$, $\Delta_j^{t+1} \leftarrow \Delta_j^t$, and $R_j^{t+1} \leftarrow R_j^t$. The result of step semantics $Step(P, D)$ is a minimum size set of non-delta tuples S, such that $S = D^0 \setminus D^t$ and $D^t = D^{t+1}$.*

The result of step semantics is then the minimum possible number of tuples that are deleted by a sequence of single rule activations. If there is more than one sequence that results in a minimum number of derived delta tuples, step semantics non-deterministically outputs one of the sets of tuples associated with one of the sequences. Step semantics has two uses: (1) simulate a subset of SQL triggers ("delete after delete") to determine the logic in which they will operate in case there is a need for each trigger to operate separately and immediately update the table from which it deleted a tuple and then evaluate whether another trigger needs to operate (similar to row-by-row semantics, but for multiple triggers), and (2) DC-like semantics can also be simulated with this semantics (see paragraph at the end of this section).

Example 3.6. *Reconsider the example in Figures 1 and 2. We demonstrate a sequence of rule activations that results in the smallest set of deleted tuples in step semantics.*

(1) At step $t = 1$, there is one satisfying assignments to rule (0) that derives $\Delta(g_2)$. We update $\Delta^1_{\texttt{Grant}} = \{g_2\}$, $\texttt{Grant}^1 = \{g_1\}$.

(2) At step $t = 2$, there are two satisfying assignments to rule (1). We choose the assignment deriving $\Delta(a_2)$, and update D^1 so that $\Delta^2_{\texttt{Author}} = \{a_2\}$, $\texttt{Author}^2 = \{a_1, a_3\}$.

(3) In step $t = 3$, we have three satisfying assignments: to rules (1), (2), and (3). Suppose we choose the one satisfying rule (1) and derive $\Delta(a_3)$. D^2 is now updated such that $\Delta^3_{\texttt{Author}} = \{a_2, a_3\}$, $\texttt{Author}^3 = \{a_1\}$.

(4) In step $t = 4$, there are two assignments to rule (2) and two to rule (3). We choose the one deriving $\Delta(w_1)$ and update D^3 with $\Delta^4_{\texttt{Writes}} = \{w_1\}$, $\texttt{Writes}^4 = \{w_2\}$. Note that in the next step, the assignment to rule (3) deriving $\Delta(p_1)$ will not be possible due to this update.

(5) In step $t = 5$, there is an assignment to rule (2) and two assignments to rule (3). We choose the one deriving $\Delta(w_2)$ and update D^4 with $\Delta^5_{\texttt{Writes}} = \{w_1, w_2\}$, $\texttt{Writes}^5 = \emptyset$.

The result for this example is depicted in Figure 4 where the gray and green tuples are deleted from the original relations and added to the delta relations.

3.4 Stage Semantics

Stage semantics separates the evaluation process into stages so at each stage we employ all satisfying assignments to derive *all possible tuples*, and update the delta relations and the original relations (after all possible tuples are found). At each stage t of evaluation (similarly to the semi-näive algorithm [4]), we compute all tuples for Δ_i relations and update the relations R_i in this stage by $R_i^t = R_i^{t-1} \setminus \Delta_i^t$.

Definition 3.7. *Let D and P be a database and a delta program over the schema $\mathbf{R} \cup \mathbf{\Delta}$, respectively. According to stage semantics, at stage $t = 0$, $\Delta_i^t = \emptyset$ and R_i^t is the relation R_i in D. For each stage $t > 0$, $\Delta_i^t \leftarrow \Delta_i^{t-1} \cup \{tup \mid tup = \alpha(head(r)), r \in P, \alpha[body(r)] \in D^{t-1}, \alpha : body(R) \rightarrow D^{t-1}\}$, and $R_i^t \leftarrow R_i^{t-1} \setminus \Delta_i^t$. The result of stage semantics, denoted $Stage(P, D)$, is the set of non-delta tuples S, such that $S = D^0 \setminus D^t$ and $D^t = D^{t+1}$.*

This semantics can be used to simulate a subset of SQL triggers to determine the logic in which they will operate in case there is a need for several stages of deletions of tuples, i.e., the triggers lead to a cascade deletion.

Example 3.8. *Reconsider the database in Figure 1 and the rules in Figure 2. Assume we want to perform cascade deletion through triggers such that a deletion of the* Author *tuple including the* Grant *tuple including ERC will delete its recipients'* Author *tuples, and the latter will result in the deletion of the associated* Writes *and* Pub *tuples. The following describes the operation of stage semantics simulating this process:*

(1) At the first stage, there is one assignments to rule (0) deriving $\Delta(g_2)$, we update $\Delta_{\texttt{Grant}} = \{g_2\}$, $\texttt{Grant} = \{g_1\}$.

(2) At the second stage, we use the two assignments to rule (1) to derive $\Delta(a_2)$ and $\Delta(a_3)$. We update the database so that $\texttt{Author} = \{a_1\}$, $\Delta_{\texttt{Author}} = \{a_2, a_3\}$.

(3) In the next stage, we use the two assignments to rule (2) and the two assignments to rule (3) to derive $\Delta(p_1)$, $\Delta(p_2)$, $\Delta(w_1)$ and $\Delta(w_2)$, and update the database as $\texttt{Writes} = \emptyset$, $\texttt{Pub} = \emptyset$, $\Delta_{\texttt{Writes}} = \{w_1, w_2\}$, $\Delta_{\texttt{Pub}} = \{p_1, p_2\}$.

For any stage > 3, the state of the database will be identical, so this is the result of stage semantics, shown in Figure 4 where

the tuples in gray, green, and pink are deleted from the original relations and added to the delta relations.

Since the delta relations are monotone and can only be as big as the base relation, we can show the following (for brevity, the formal proofs are deferred to the full version).

Proposition 3.9. *Let* $\mathbf{R}$ *be a relational schema. For every database and delta program over* $\mathbf{R}$*, stage semantics will converge to a unique fixpoint.*

Proof Sketch. As stage semantics is rule-order independent and deterministic, at stage t we add all the Δ_i tuples that can be derived from D^t to get Δ_i^{t+1}, and further delete all the tuples in Δ_i^{t+1} from R_i^t to get R_i^{t+1}. Furthermore, the number of tuples with relations in $\mathbf{R}$ is monotonically decreasing. Thus, there exists a stage in which no more tuples with these relations who satisfy the rules exist. This is the stage that defines the fixpoint. □

3.5 End Semantics

Finally, as a baseline, we define end semantics following standard datalog evaluation of delta relations.

Definition 3.10. *Let D and P be a database and a delta program over the schema $\mathbf{R} \cup \mathbf{\Delta}$. For $t = 0$, we have $\Delta_i^t = \emptyset$ and R_i^t is the relation R in D. According to end semantics, at each state $t > 0$, $R_i^t \leftarrow R_i^0$, and $\Delta_i^t \leftarrow \Delta_i^{t-1} \cup \{tup \mid tup = \alpha(head(r)), r \in P, \alpha[body(r)] \in D^{t-1}, \alpha : body(R) \rightarrow D^{t-1}\}$. Denote the fixpoint of this semantics as T, i.e., $D^T = D^{T+1}$ At state T, $R_i^T \leftarrow R_i^0 \setminus \Delta_i^{T-1}$, $\Delta_i^T \leftarrow \Delta_i^{T-1}$. The result of end semantics $End(P, D)$ is the set of non-delta tuples $S = D^0 \setminus D^T$.*

This is the standard datalog semantics in the sense that it treats the relations in $\mathbf{\Delta}$ as regular intensional relations and only updates them during the evaluation. Once the evaluation process is completed, the relations in $\mathbf{R}$ are updated.

Example 3.11. *For the database and rules in Figures 1 and 2, all possible delta tuples will be derived using the rules as shown in Example 2.1, i.e,* $\{\Delta(\mathsf{g}_2), \Delta(\mathsf{a}_2), \Delta(\mathsf{a}_3), \Delta(\mathsf{w}_1), \Delta(\mathsf{w}_2), \Delta(\mathsf{p}_1), \Delta(\mathsf{p}_2), \Delta(\mathsf{c})\}$*. Then, after the derivation process is done, the tuples* $\{\mathsf{g}_2, \mathsf{a}_2, \mathsf{a}_3, \mathsf{w}_1, \mathsf{w}_2, \mathsf{p}_1, \mathsf{p}_2, \mathsf{c}\}$ *will be deleted, to get the database appearing in Figure 4 where the gray, green, pink and orange colored tuples are deleted from the original relations and added to the delta relations.*

As end semantics is closely related to datalog evaluation, it inherits the basic property of converging to a unique fixpoint.

3.6 Stabilizing Sets and Problem Statement

After defining delta programs, we introduce the notion of a *stable database* with respect to a delta program.

Definition 3.12. *Given a database D over a schema $\mathbf{R} \cup \mathbf{\Delta}$, and a delta program P, D is a stable database w.r.t P if $\{\alpha(head(r)) \mid r \in P, \alpha : body(r) \rightarrow D, \alpha(body(r)) \in D\} = \emptyset$, i.e., D does not satisfy any rule in P.*

Example 3.13. *Reconsider the database in Figure 1 and the rules in Figure 2. If we remove the tuples included in the result of end semantics in Example 3.11 and add their corresponding delta tuples, we would have a stable database.*

Alternatively, we can say that a stable database w.r.t. a delta program is a database where no delta tuples can be generated. A database is said to be unstable if it is not stable.

Definition 3.14. *Given an unstable database D w.r.t a delta program P over a schema $\mathbf{R} \cup \mathbf{\Delta}$, a stabilizing set for D is a set of tuples S such that $(D \setminus S) \cup \Delta(S)$ is stable.*

Example 3.15. *Returning to Example 3.13, a stabilizing set would be* $S = \{\mathsf{g}_2, \mathsf{a}_2, \mathsf{a}_3, \mathsf{w}_1, \mathsf{w}_2, \mathsf{p}_1, \mathsf{p}_2, \mathsf{c}\}$*, as the database without these tuples and with the tuples in $\Delta(S)$ does not satisfy any of the rules in Figure 2.*

Our objective is to study the complexity and the relationships between the semantics we have defined, and devise efficient algorithms to find their solutions.

Definition 3.16 (Problem Definition). *Given (D, P, σ), where D is a database and P is a delta program over schema $\mathbf{R}, \mathbf{\Delta}$, and σ is a semantics, the desired solution is the result of σ w.r.t. D and P, denoted by $\sigma(D, P)$.*

Initialization of the database and the deletion process. The deletion process can start in two ways. When the given database contains tuples that violate the constraints expressed by the delta program. This is a popular scenario for data repair. Another scenario is where the initial database is stable and the user wants to delete a specific set of tuples. At start, we assume $\Delta_i = \emptyset$ for all i. To start the deletion process, we add a rule for each tuple $R_i(\mathbf{C})$ of the form $\Delta_i(\mathbf{C}) :- R_i(\mathbf{C})$.

Example 3.17. *Consider a slightly different schema than the database in Figure 1 where the* Pub *table also mentions the conference in which each paper was published and the delta rule* $\Delta_{\mathsf{Pub}}(p_1, t_1, conf_1) : -\mathsf{Pub}(p_1, t_1, conf_1), \mathsf{Pub}(p_2, t_1, conf_2)$ *stating that no two papers with the same title can be in published in two two different conferences. An unstable database with two tuples* $\mathsf{Pub}(1, X, C_1)$ *and* $\mathsf{Pub}(1, X, C_2)$ *will violate this rule and start the deletion process. In our running example, however, we would like to start the deletion process by deleting the tuple* g_2*, and for this we have defined rule (0) in Figure 2.*

We can observe the following:

Proposition 3.18. *Given a database D, a delta program P, and a semantics σ, both D and $\sigma(P, D)$ are always stabilizing sets under all four semantics, where $\sigma(P, D)$ is the result of σ given P and D. In other words, a stabilizing set always exists.*

Intuitively, if the database is stable, a stabilizing set is the empty set. Otherwise, the entire database is a stabilizing set. Additionally, the result of each semantics is defined as the set of non-delta tuples S such that $(D \setminus S) \cup \Delta(S)$ is stable. Note that sometimes these sets and the results of the different semantics are identical. E.g., if there is only one tuple in the database and one delta rule that deletes it, then this tuple forms the unique stabilizing set and will be returned by all semantics. Moreover, the results of independent and step semantics may not be unique:

Proposition 3.19. *There exist a database D and a delta program P such that there are two possible results for independent and step semantics.*

To see this, consider the database $D = \{R_1(a), R_2(b)\}$ and a program with two rules (1) $\Delta_1(x) : -R_1(x), R_2(y)$, and (2) $\Delta_2(y) : -R_1(x), R_2(y)$. For independent and step semantics, there are two equivalent solutions: $\{R_1(a)\}$ derived from rule (1), or $\{R_2(b)\}$ derived from rule (2).

Expressiveness of delta rules. We discuss some forms of constraints that are captured by delta rules. DCs [10] can be written as a first order logic statement: $\forall \mathbf{x_1}, \ldots, \mathbf{x_m} \quad \neg(R_1(\mathbf{x_1}), \ldots, R_m(\mathbf{x_m}), \varphi(\mathbf{x_1}, \ldots, \mathbf{x_m}))$. $\varphi(\mathbf{x_1}, \ldots, \mathbf{x_m})$ is a conjunction of atomic formulas of the form $R_i[A_k] \circ R_j[A_l]$, $R_i[A_k] \circ \alpha$, where α is a constant, and $\circ \in \{<, >, =, \neq, \leq, \geq\}$. Given a DC, C, of this form, we translate it to the following delta rule:

$$\Delta_1(\mathbf{x_1}) : -R_1(\mathbf{x_1}), \ldots, R_m(\mathbf{x_m}), \{A_k^i \circ A_l^j \mid R_i[A_k] \circ R_j[A_l] \in C\}, \{A_k^i \circ \alpha \mid R_i[A_k] \circ \alpha \in C\}$$

The first part of the body contains the atoms used in C, the second part contains the comparison between different attributes in C and the third contains the comparison between a attribute and a constant in C. For independent semantics, the head of the rule can be any delta atom $\Delta_i(\mathbf{x_i})$. $Ind(P, D)$ will then be the smallest set of tuples that should be deleted such that the rule is not satisfied, i.e., from each set of tuples that violate C, at least one tuple will be deleted. I.e., $Ind(P, D)$ will be the smallest set of tuples that needs to be deleted such that the database will comply with C. Step semantics can also mimic this by adding a rule for each atom in the rule corresponding to the C. We will have m rules and each will have as a head one of the atoms participating in the DC. Thus, for each set of of tuples violating C, we have a set of m rules allowing us to delete any tuple from this set. Note that in both $Ind(P, D)$ and $Step(P, D)$, only one tuple from the violating set would be deleted.

Similarly, we can show that delta rules, along with the appropriate semantics, can express Domain Constraints [16], "after delete, delete" SQL Triggers [22], and Causal Rules without recursion [46] (whose syntax inspired delta rules).

We now compare the results obtained from the semantics in terms of set containment and size.

Proposition 3.20. *Given database D and delta program P,*

(1) $|Ind(P, D)| \leq |Step(P, D)|, |Stage(P, D)|$, and there is a case where $|Ind(P, D)| < |Step(P, D)|, |Stage(P, D)|$

(2) $Stage(P, D) \subseteq End(P, D)$, and there is a case where $Stage(P, D) \subsetneq End(P, D)$

(3) $Step(P, D) \subseteq End(P, D)$, and there is a case where $Step(P, D) \subsetneq End(P, D)$

(4) There exists cases where $Step(P, D) \subsetneq Stage(P, D)$ and cases where $Stage(P, D) \subsetneq Step(P, D)$

4 COMPLEXITY OF FINDING RESULTS

We now analyze the complexity for each semantics.

End semantics. We follow datalog-like semantics, so the stabilizing set according to end semantics is unique and defined by the single fixpoint. Therefore, we can utilize the standard datalog semantics, treating relations in $\mathbf{\Delta}$ as intensional and deriving all possible delta tuples from the program. After the evaluation is done, we update the relations in $\mathbf{R}$ by removing from them the delta tuples that have been derived.

Stage semantics. Similar to end semantics, for stage semantics, if we evaluate the program over the database, we would arrive at a fixpoint. Here, we apply a different evaluation technique, separating the evaluation into stages. At each stage of evaluation, we derive all possible tuples through satisfied rules, and update the database. We continue in this manner until no more tuples can be derived.

Proposition 4.1. *Given a database D and delta program P, computing $End(P, D)$, $Stage(P, D)$ is PTime in data complexity.*

Independent and step semantics. Unlike end and stage semantics, the other two semantics are computationally hard:

Proposition 4.2. *Given a delta program P, an unstable database D w.r.t P, and an integer k, it is NP-hard in the value of k to decide whether $|Ind(P, D)| \leq k$ or $|Step(P, D)| \leq k$.*

proof sketch. We reduce the decision problem of minimum vertex cover to finding $Step(P, D)$ and $Ind(P, D)$. Given a graph $G = (V, E)$ and an integer k, we define an unstable database D: for every $(u, v) \in E(G)$ we have $E(u, v), E(v, u) \in D$ and for every $v \in V(G)$ we have $VC(v) \in D$. For independent semantics we define the delta program: (1) $\Delta_{VC}(x)$: $-E(x, y), VC(x), VC(y)$, (2) $\Delta_{VC}(x)$: $-VC(x), \Delta_E(x, y)$, (3) $\Delta_{VC}(y)$: $-VC(y), \Delta_E(x, y)$. For step semantics, we only need rule (1). Rules (2) and (3) are only used in the reduction to independent semantics to make the derivation of tuples of the form $E(a, b)$ not worthwhile (as in this semantics, tuples can be removed from E and added to Δ_E without being derived). We can show that a vertex cover of size $\leq k$ is equivalent to $|Ind(P, D)| \leq k$ with the first program and $|Step(P, D)| \leq k$ with the second program. □

Naturally, if we consider the search problem, k is unknown and, in the worst case, may be the size of the entire database.

Algorithm 1: Find Stabilizing Set - Independent

Input : Delta program P, unstable database D
Output: A stabilizing set $S \subseteq D$

1 Consider all possible tuples in $t \in D \cup \Delta(D)$ and store the DNF provenance for each tuple t;
2 Let F be an empty Boolean formula;
3 **foreach** $t \in P(D)$ **do**
4 $\quad F \leftarrow F \vee Prov(t)$;
5 $\alpha \leftarrow$ Min-Ones-SAT($\neg F$);
6 output $\{t' \mid \alpha(\neg t') = True\}$;

5 HANDLING INTRACTABLE CASES

We now present algorithms to handle independent and step semantics, which are NP-hard by Proposition 4.2.

5.1 Algorithm for Independent Semantics

Our approach relies on the provenance represented as a Boolean formula [26], where the provenance of each tuple is a DNF formula, each clause describing a single assignment and delta tuples are negated variables.

Algorithm 1 uses this idea to find a stabilizing set. We generate the provenance of each *possible* delta tuple (not only for the ones that can be derived using an operational semantics and the rules) represented as a Boolean formula (line 1). This is a DNF formula for each delta tuple, where tuples with relations in **R** are represented as their own literals and tuples in with relations in **Δ** are represented as the negation of their counterpart tuples with relations in **R**. In lines 2–4 we connect these formulae using $\vee$ into one formula representing the provenance of all the delta tuples (this is a disjunction of DNFs). We negate this formula, resulting in a conjunction of CNFs. We then find a *satisfying assignment giving a minimum number of True values to negated variables.* In the negated formula, each satisfied clause says that at least one of the tuples needed for the assignment the clause represents is not present in the database. An assignment that gives the minimum number of negated variables the value True represents the minimum number of tuples whose deletion from the database and addition of their delta counterparts would stabilize the database. Changing negated variables to positive ones and vice-versa will give us an instance of the *min-ones SAT* problem [31] (line 5), where the goal is to find a satisfying assignment to a Boolean formula, which maps the minimum number of variables to $True$. In line 6, we output the facts whose negated form is mapped to True.

Example 5.1. *Reconsider the database in Figure 1 and the program composed of the rules in Figure 2. Algorithm 1 generates the provenance formula and negates it:*

$$\neg \mathsf{g}_2 \wedge (\neg \mathsf{a}_2 \vee \neg \mathsf{ag}_2 \vee \mathsf{g}_2) \wedge (\neg \mathsf{a}_3 \vee \neg \mathsf{ag}_3 \vee \mathsf{g}_2) \wedge (\neg \mathsf{p}_1 \vee \neg \mathsf{w}_1 \vee \mathsf{a}_2) \wedge (\neg \mathsf{p}_2 \vee \neg \mathsf{w}_2 \vee \mathsf{a}_3) \wedge (\neg \mathsf{c} \vee \mathsf{p}_1 \vee \neg \mathsf{w}_1 \vee \neg \mathsf{w}_2)$$

It then generates the assignment giving the value True to the smallest number of negated literals in line 5. This satisfying assignment is α such that $\alpha(\mathsf{g}_2) = \alpha(\mathsf{ag}_2) = \alpha(\mathsf{ag}_3) = False$ and gives every other variable the value True. Finally, in line 6, the algorithm returns the set of tuples that α mapped to False, i.e., $\{\mathsf{g}_2, \mathsf{ag}_2, \mathsf{ag}_3\}$, as in Example 3.4.

Correctness: If procedure min-ones SAT finds the minimum satisfying assignment, Algorithm 1 outputs $Ind(P, D)$. Yet, any satisfying assignment would form a stabilizing set.
Complexity: Given a database D and a program P, the complexity of computing the provenance Boolean formula is $|D|^{O(|P|)}$; the time to use a solver to find the minimum satisfying assignment is theoretically not polynomial, however, such algorithms are efficient in practice.

5.2 Algorithm for Step Semantics

We describe a greedy algorithm (Algorithm 2) for step semantics. We will use the concepts of *provenance graph* and the *benefit* of a tuple. A provenance graph [18] is a collection of derivation trees [4]. A derivation tree of a tuple, $T = (V, E)$, illustrates the tuples that participated in its derivation (the set of nodes V), and the rules that were used [17] (each rule that uses $t_1, \ldots, t_k$ to derive t is modeled by edges from $t_1, \ldots, t_k$ to t). When there are several derived tuples of interest, a provenance graph joins together derivation trees, but the input tuples appear only once and are reused in the graph. In our case, only delta tuples are derived, so we define the provenance graph as follows: each tuple is associated with a node and there is an edge from t_1 to $\Delta(t_2)$ if t_1 participates in an assignment resulting in $\Delta(t_2)$. The *benefit* of each non-delta node t is the number of assignments it participates in minus the number of assignments $\Delta(t)$ participates in. Therefore, the benefit of each non-delta node t equals the number of its outgoing edges minus the number of outgoing edges from $\Delta(t)$. Algorithm 2 stores the provenance for all delta tuples generated by end semantics as a graph. For each leaf node in this graph for input tuples t, we store its benefit b_t.

Intuitively, Algorithm 2 chooses for the output stabilizing set only tuples that can be derived using the delta rules and prioritizes at each layer (using the benefits) those tuples t where the number of assignments eliminated by deletion of t is large, and the number of assignments enabled by the creation of $\Delta(t)$ is small.

We consider the nodes of the provenance graph G in each layer and the set of assignments *Assign*. For each layer i in G, we greedily choose to add to the stabilizing set the tuple t, where $\Delta(t)$ is in layer i (layer i is denoted by G_i) and b_t is the maximum across all tuples t where $\Delta(t)$ is in layer i. We then delete the subgraph induced by $\{\Delta(t') \mid \forall \alpha \in Assign \text{ s.t. } Im(\alpha) = \Delta(t') \; \exists t_k \in Dom(\alpha) \cap S \wedge t' \neq t_k\}$. In words, we delete all delta tuples, such that each one of their

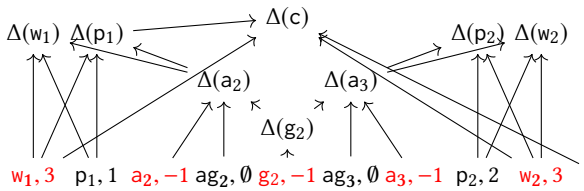


Figure 5: Provenance graph for D in Figure 1 and the program in Figure 2. Red tuples are chosen for the set returned by Algorithm 2

Algorithm 2: Find Stabilizing Set - Step

Input : Delta program P, unstable database D
Output: A stabilizing set $S \subseteq D$

```
Store the directed provenance graph G of End(P, D);
Compute b_t for each non-delta tuple t;
Assign ← {α | α is an assignment that derives Δ(t) ∈ Δ(End(P, D))};
S ← ∅;
foreach Layer 1 ≤ i ≤ L do
    while ∃Δ(t) ∈ G_i s.t. t ∉ S do
        t_m = arg max_{t∈G, Δ(t)∈G_i} b_t;
        S ← S ∪ {t_m};
        G ← G \ G[Δ(t') | ∀α ∈ Assign s.t. Im(α) = Δ(t') ∃t_k ∈ Dom(α) ∩ S ∧ t' ≠ t_k];
output S;
```

assignments contains a tuple t_k that was chosen to be deleted, except $\Delta(t_k)$ itself and the tuples reachable from it, since adding t_k to S implies that $\Delta(t_k)$ has been generated and can participate in other derivation. We continue this process until only the delta counterparts of the selected tuples remain in the provenance graph. This ensures that we only delete delta tuples that cannot be generated by any assignment.

Example 5.2. *Reconsider our running example. Its provenance graph according to end semantics is shown in Figure 5. After computing b_t for all the leaf tuples, we begin iterating over the layers of the graph. In layer 1 we only have $\Delta(g_2)$, with $b_{g_2} = -1$, so we choose it. Since g_2 is only connected to $\Delta(g_2)$, we do not change G. We then continue to layer 2 where we have $\Delta(a_2)$ and $\Delta(a_3)$. We arbitrarily choose a_2 as $b_{a_2} = b_{a_3} = -1$, and do not change G. After that, we choose a_3 and again not change G. In layer 3, we have w_1, w_2, p_1, p_2 where $b_{p_2} < b_{p_1} < b_{w_1} = b_{w_2}$, so we choose arbitrarily to include w_1 in S. We then delete from G the subgraph induced by $\Delta(w_1)$. Since there are more delta tuples in this layer we continue to choose w_2 and delete from G the subgraph induced by $\Delta(w_2)$. Since there are no more delta tuples in layers 3 and 4 except $\Delta(w_1), \Delta(w_2)$ where $w_1, w_2 \in S$, we return $S = \{g_2, a_2, a_3, w_1, w_2\}$.*

Correctness: Algorithm 2 returns a stabilizing set that can be derived using the delta rules; the minimum set it can return is $Step(P, D)$, but in general provides a heuristic.

Complexity: Given a database D and a program P, the overall complexity of Algorithm 2 is $|D|^{O(|P|)}$, since it is the size of the provenance graph.

6 IMPLEMENTATION & EXPERIMENTS

We have implemented our algorithms in Python 3.6 with the underlying database stored in PostgreSQL 10.6. Delta rules are implemented as SQL queries and delta relations are auxiliary relations in the database. For Algorithm 1 we have used the Z3 SMT solver [15] and specifically, the relevant part that allows for the formulation of optimization problems such as Min-Ones-SAT [7], which draws on previous work in this field [32, 41, 47]. For Algorithm 2, we have used Python's NetworkX package [27] to model the graph and manipulate it as required by the algorithm. The approach used to evaluate the results of all semantics is a standard näive evaluation, evaluating all rules iteratively, and terminating when no new tuples have been generated. The experiments were performed on Windows 10, 64-bit, with 8GB of RAM and Intel Core Duo i7 2.59 GHz processor, except for the `HoloClean` comparison which was performed on Ubuntu 18 on a VMware workstation 12 with 6.5GB RAM allotted. The reason for that is that the Torch package version 1.0.1.post2 required for `HoloClean` did not run on Windows.

Databases: We have used a fragment of the MAS database [35], containing academic information about universities, authors, and publications. It includes over 124K tuples and the following relations: Organization(oid, name), Author(aid, name, oid), Writes(aid, pid), Publication(pid, title, year), Cite(citing, cited). We have also used a fragment of the TPC-H dataset [50], which included 376,175 tuples. This dataset includes 8 tables (customer, supplier, partsupp, part, lineitem, orders, nation, and region).

Test programs: Tables 1 and 2 show the programs we have used for the MAS and TPC-H datasets experiments, respectively. We use the first letter of each table as an abbreviation, and denote by C/C_i a constant we have assigned to an attribute. The programs were designed for different scenarios to compare the four semantics and highlight the manner in which each semantics is advantageous. The programs can roughly be divided into three sets: (1) those that are meant to mimic the semantics of integrity constraints such as DCs (programs 1–4, 11–15 in Table 1), (2) those that are meant to perform cascade deletion (programs 5, 9, 10, and 16–20 in Table 1 and programs 1–3 in Table 2), and (3) those that mix between the two (programs 6–8 in Table 1, and programs 4–6 in Table 2). For programs that express integrity constraints, independent semantics would guarantee a minimum size repair while the other semantics may delete a larger number of tuples. For example, in program 2 in Table 1, using end, stage or step semantics may yield a result

Table 1: MAS Programs

Num.	Program
1	(1) $\Delta_A(aid, n, oid) : -A(aid, n, oid), n = C_1$ (2) $\Delta_W(aid, pid) : -W(aid, pid), aid = C_2$
2	(1) $\Delta_W(aid, pid) : -W(aid, pid), A(aid, n, oid), aid = C$
3	(1) $\Delta_A(aid, n, oid) : -W(aid, pid), A(aid, n, oid), aid = C$ (2) $\Delta_W(aid, pid) : -W(aid, pid), A(aid, n, oid), aid = C$
4	(1) $\Delta_A(aid, pid) : -O(oid, n_2), A(aid, n, oid), oid = C$ (2) $\Delta_O(aid, pid) : -O(oid, n_2), A(aid, n, oid), oid = C$
5	(1) $\Delta_A(aid, n, oid) : -A(aid, n, oid), n = C_1$ (2) $\Delta_W(aid, pid) : -W(aid, pid), \Delta_A(aid, n, oid)$
6	(1) $\Delta_A(aid, n, oid) : -A(aid, n, oid), n = C_1$ (2) $\Delta_W(aid, pid) : -W(aid, pid), \Delta_A(aid, n, oid)$ (3) $\Delta_P(pid, t) : -P(pid, t), \Delta_W(aid, pid), A(aid, n, oid)$
7	(1) $\Delta_P(pid, t) : -P(pid, t), pid = C$ (2) $\Delta_C(pid, cited) : -C(pid, cited), \Delta_P(pid, t)$ (3) $\Delta_C(citing, pid) : -C(citing, pid), \Delta_P(pid, t)$
8	(1) $\Delta_A(aid, n, oid) : -W(aid, pid), A(aid, n, oid), aid = C$ (2) $\Delta_W(aid, pid) : -W(aid, pid), A(aid, n, oid), aid = C$ (3) $\Delta_P(pid, t) : -P(pid, t), \Delta_W(aid, pid), A(aid, n, oid)$ (4) $\Delta_P(pid, t) : -P(pid, t), W(aid, pid), \Delta_A(aid, n, oid)$
9	(1) $\Delta_A(aid, n, oid) : -A(aid, n, oid), n = C$ (2) $\Delta_W(aid, pid) : -W(aid, pid), \Delta_A(aid, n, oid)$ (3) $\Delta_P(pid, t) : -P(pid, t), \Delta_W(aid, pid)$ (4) $\Delta_C(pid, cited) : -C(pid, cited), \Delta_P(pid, t), pid < C$
10	(1) $\Delta_O(oid, n_2) : -O(oid, n_2), oid = C$ (2) $\Delta_A(aid, n, oid) : -A(aid, n, oid), \Delta_O(oid, n_2)$ (3) $\Delta_W(aid, pid) : -W(aid, pid), \Delta_A(aid, n, oid)$ (4) $\Delta_P(pid, t) : -P(pid, t), \Delta_W(aid, pid)$
11–15	$\Delta_C(pid, c_2) : -\{\{\{\{C(pid, c_2)\}^{11}, P(t, pid)\}^{12}, W(aid, pid)\}^{13}, A(aid, n, oid)\}^{14}, O(oid, n_2)\}^{15}$
16–20	(1) $\Delta_O(oid, n_2) : -O(oid, n_2), oid = C$ (prog. 16-20) (2) $\Delta_A(aid, n, oid) : -A(aid, n, oid), \Delta_O(oid, n_2)$ (prog. 17-20) (3) $\Delta_W(aid, pid) : -W(aid, pid), \Delta_A(aid, n, oid)$ (prog. 18-20) (4) $\Delta_P(pid, t) : -P(pid, t), \Delta_W(aid, pid)$ (prog. 16-20) (5) $\Delta_C(citing, pid) : -C(citing, pid), \Delta_P(pid, t)$ (prog. 20)

Table 2: TPC-H Programs

Num.	Program
1	(1) $\Delta_{PS}(sk, X) : -PS(sk, X), S(sk, Y), sk < C$ (2) $\Delta_{LI}(sk, X) : -LI(sk, X), \Delta_{PS}(sk, Y)$
2	(1) $\Delta_{PS}(sk, X) : -PS(sk, X), sk < C$ (2) $\Delta_{LI}(sk, X) : -LI(sk, X), \Delta_{PS}(sk, Y)$
3	(1) $\Delta_{PS}(sk, pk, X) : -PS(sk, pk, X), S(sk, Y), P(pk, Y), sk < C$ (2) $\Delta_{LI}(sk, X) : -LI(sk, X), \Delta_{PS}(sk, Y)$
4	(1) $\Delta_{LI}(ok, X) : -LI(ok, X), ok < C_2$ (2) $\Delta_S(sk, X) : -S(sk, X), \Delta_{LI}(sk, ok, Y)$ (3) $\Delta_C(ck, X) : -C(ck, X), O(ok, ck, Y), \Delta_{LI}(ok, Z)$
5	(1) $\Delta_N(nk, X) : -N(nk, X), nk = C_3$ (2) $\Delta_S(nk, X) : -S(nk, X), \Delta_N(nk, Y), C(nk, Z)$ (3) $\Delta_C(nk, X) : -S(nk, X), \Delta_N(nk, Y), C(nk, Z)$
6	(1) $\Delta_O(ck, X) : -O(ck, X), C(ck, Y), ck < C_4$ (2) $\Delta_{PS}(sk, X) : -PS(sk, X), S(sk, Y), sk < C_4$ (3) $\Delta_{LI}(sk, X) : -LI(ok, X), \Delta_O(ok, Y)$ (4) $\Delta_{LI}(sk, X) : -LI(sk, X), \Delta_{PS}(sk, Y)$

composed of Writes tuples which will likely not be minimal in size. If instead we use independent semantics, we could have a result of a single `Author` tuple. For programs that are purely designed for cascade deletion, we expect the result of all semantics to be the same and therefore the fastest and most accurate algorithm should be used, i.e., end or stage semantics. For the programs that perform a mix of the two options, it would depend on the desired result. For example, program 8 in Table 1 is designed to distinguish between stage and step semantics, where stage semantics will not be able to use rules 3 and 4, while step semantics will not be able to derive all delta tuples from both rules 1 and 2.

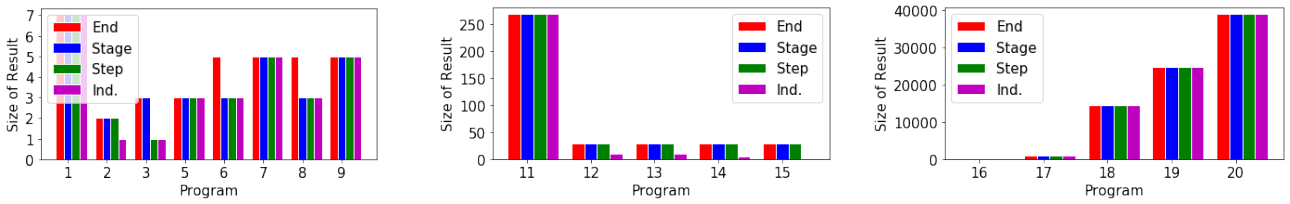
Setting and highlights: We have focused on four different aspects in our experimental study: (1) the relationship between the sets found for each semantics; (2) the size of the result set computed by each algorithm; (3) the algorithms

Table 3: Containment of results for the programs in Tables 1 and 2

Program	Step = Stage	Ind ⊆ Stage	Ind ⊆ Step
1	✓	✓	✓
2	✓	✗	✗
3	✗	✓	✓
4	✗	✓	✓
5	✓	✓	✓
6	✓	✓	✓
7	✓	✓	✓
8	✗	✗	✓
9	✓	✓	✓
10	✓	✓	✓
11	✓	✓	✓
12–15	✓	✗	✗
16–20	✓	✓	✓
T-1	✓	✗	✗
T-2	✓	✓	✓
T-3	✓	✗	✗
T-4	✓	✗	✗
T-5	✗	✓	✓
T-6	✓	✗	✗

execution times and their breakdown and (4) a comparison of our approach with PostgrSQL and MySQL triggers, and a comparison with the state-of-the-art data repair system `HoloClean` [44] that repairs cells instead of deleting tuples. We have manually checked that Algorithms 1, 2 output the actual result for programs 1, 2, 3, 5–9 (where the sizes of the result are small enough to be manually verified). Hence, we refer to the output given by these algorithms as the result of the two semantics. All of the algorithms computed the results in feasible time (the average runtimes for end, stage, step and independent were 16.9, 21.1, 389.5, and 73 seconds resp. for the programs in Table 1). In general, computing the results of end and stage semantics is faster than those of step and stage semantics. Thus, for programs that perform cascade deletion (e.g., 16–20 in Table 1), where the result for all semantics is the same, it may be preferable to use end or stage semantics. For programs such as 11–15 in Table 1, where there is a clear difference between the results, users may choose the desired semantics they wish to enforce, while aware of the difference in performance. As an example, for these programs, independent semantics would correspond to the semantics of DCs (but would be slower to compute the repair), while the other ones would correspond to triggers. We also demonstrate the discrepancy between the results of the different semantics using specific programs and Table 3 showing the relationships between the results. For example, for program 8 in Table 1, there is a no containment of the result of stage in the result of step semantics and vice versa.

Containment of results: Table 3 shows the relationship between the results generated for the different semantics. The table has three columns: $Step = Stage$, describing whether the result of stage semantics is equal to the result of step semantics, $Ind \subseteq Stage$ and $Ind \subseteq Step$ which capture whether the result of independent semantics is contained in the result of stage and step semantics respectively. The other



(a) Size of results; Programs 1–10 (b) Size of results; Programs 11–15 (c) Size of results; Programs 16–20

Figure 6: Comparison of result sizes for the four semantics with the programs from Table 1 (prog. 4, 10 in text)

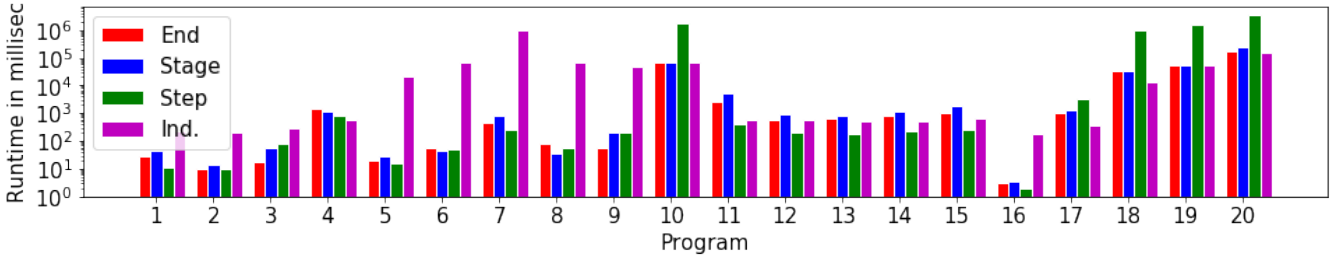
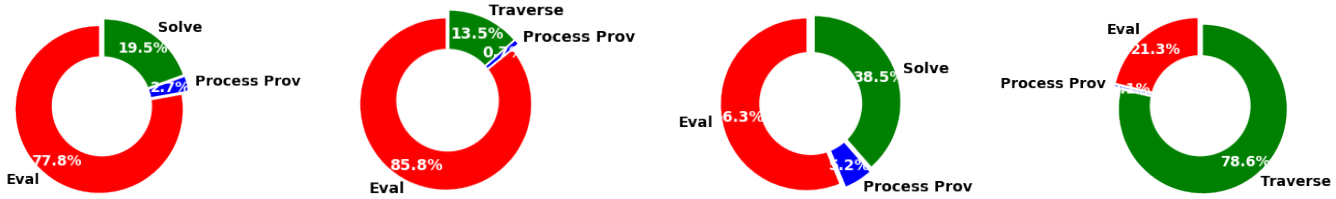
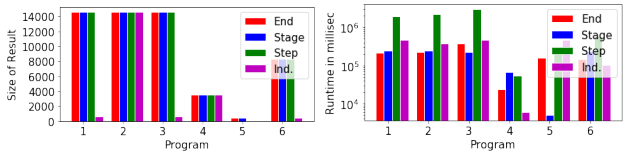


Figure 7: Execution time for finding the results of the four semantics with the programs from Table 1



(a) Algo. 1 (1–15) (b) Algo. 2 (1–15) (c) Algo. 1 (16–20) (d) Algo. 2 (16–20)

Figure 8: Runtime breakdown for programs 1–15 and 16–20, and Algorithms 1 (ind. sem.) and 2 (step sem.)



(a) Size of results; TPC-H (b) Runtime; TPC-H

Figure 9: Comparison of results sizes and runtimes for the four semantics with TPC-H programs

relationships always hold, as shown in Figure 3. We start by reviewing the results for the programs in Table 1. For program 2, there is no containment of the result of independent semantics, since it includes a single Author tuple which cannot be derived, so it cannot be in the results of stage or step semantics. Programs 3 and 4 are composed of two rules with the same body, so the result of stage semantics contains all derivable tuples while the result of step and independent semantics contains only one Author tuple (this is also evident in Figure 6a for program 3). Program 8 was designed based on the proof of Proposition 3.20, and thus "separates" between step and stage semantics. For programs 12–15, the tuples chosen for the result in independent semantics cannot be derived and hence there is no containment. Finally, for programs 16–20, all derived tuples have to be included in the result, according to all semantics and, therefore, all the conditions in the table are true. The results for the TPC-H programs in Table 2 are shown in the lower part of Table 3 with the prefix "T". As for the first column, we found that only for program 5, $Stage \not\subseteq Step$. This program contains two rules with the same body, and step semantics was able to delete fewer tuples by selecting the minimal set of Customer and Supplier delta tuples to derive. For the second and third columns, the result of independent semantics was not contained in the result of either step or stage or both for all programs except programs 2 and 5, as Algorithm 1 deleted tuples that were not derivable by other semantics.

Results size: Figure 6 depicts the results size for the different programs in Table 1. For the chart in Figure 6a, we included all programs except for 4 and 10, as they would have distorted the scale. For program 4, the sizes were 956 for end and stage semantics and 1 for step and independent semantics. For program 10, the sizes of all results were 24,798. In Figure 6a, as predicted in Figure 3, the size of the result of end semantics is always larger than the sizes of the results according to the other semantics. For program 2, the result of independent semantics can be of size 1 (the Author tuple with $aid = C$), whereas all other semantics may include

only Writes tuples, since Author tuples cannot be derived. Furthermore, note that programs 3 and 4 was designed to have only one tuple in the result of step and independent semantics (the Organization tuple with oid C), and all Author tuples along with the Organization tuple for end and stage semantics. Figure 6b shows the results for programs 11–15. Note that the results of all semantics except for independent semantics can only include Cite tuples. Thus, the results size according to end, stage and step semantics is identical for all programs, but the result size for the independent semantics actually decreases as the number of joins increases. In Figure 6c, all results sizes are equal for every program since all possible tuples need to be included in the stabilizing set by all semantics. The maximum result size for program 20 was 38,954. Figure 9a shows the sizes of the results for the TPC-H programs in Table 2, the largest being 14,550 tuples for programs 1, 2, 3 through end, stage and step semantics. The rational for the results here is similar, where for programs 1, 3, 5 and 6 Algorithm 1 (ind. semantics) outputted a smaller result by choosing tuples that were not derived by the rules.

Execution times: We have examined the execution time for the algorithms of the four semantics and all programs in Table 1 (Figure 7) and Table 2 (Figure 9b). The recorded times are presented in log scale. When the execution time is not negligible, Algorithms 1 and 2 require the largest execution time for most programs due to the overhead of generating the Boolean formula and finding the minimum satisfying assignment or generating the provenance graph and traversing it. For programs 10 and 16–20, all derived tuples participate in the result of each semantics and, hence, all algorithms have to "work hard". In particular, Algorithm 2 has to traverse a provenance graph of 5 layers for program 20. The results for Programs 11–15 (single rule with an increasing number of joins) were all fast (the slowest time was 5.5 seconds, incurred for stage semantics). Thus, an increase in the number of joins does not necessarily reflect an increase in execution time. Most computations were dominated either by Algorithm 1 or 2 as both are algorithms that store and process the provenance as opposed to the two other algorithms for end and stage semantics. In some cases, Algorithm 2 is faster than the algorithms for stage and end semantics. This happens when the runtimes are either very small (e.g., programs 1 and 2), or for programs 11–15. In the latter, stage and end semantics have to delete all tuples that are derived through the rule and add their delta counterparts to the database throughout the evaluation process. For Algorithm 2, after creating the graph, we need to traverse a single layer.

Runtime breakdown for Algorithms 1 and 2: Figure 8 shows the breakdown of the execution time for both algorithms. We have computed the average distribution of execution time across programs 1–15 and programs 16–20 in Table 1. In Figure 8a, most of the computation time is devoted to the evaluation and storage of the provenance (Eval). The second most expensive phase is finding the minimum satisfying assignment for the Boolean formula in the SAT solver (Solve). Converting the provenance to a Boolean formula does not require much time (Process Prov). Similarly, for Algorithm 2 in Figure 8b, most of the time is spent on evaluation and provenance storing (Eval). Traversing and choosing the nodes with maximum benefit is the second most expensive phase (Traverse) and finally, converting the provenance into a graph and determining the benefits is negligible (Process Prov). Figure 8c shows the breakdown for programs 16–20. Algorithm 1 devotes a larger percentage to solving the Boolean formula. Figure 8d shows that most of the execution time is devoted to traversing the provenance graph and finding the tuples to include in the outputted set.

Comparison with Triggers: Triggers [22, 39, 49] is a standard approach for updating the database when constraints are violated. We have implemented Programs 3, 4, 5, 8 and 20 from Table 1 using triggers both in PostgreSQL and in MySQL. For programs 3 and 4, where two triggers are programmed to fire at the same event, the PostgreSQL triggers were fired alphabetically by their assigned name while the MySQL triggers fired by the order in which they were written. Due to this fact, for program 4, the PostgreSQL triggers deleted all Author tuples associated with a single organization, instead of one Organization tuple. In these scenarios, using step semantics would have yielded a smaller result. Both PostgreSQL and MySQL triggers have led to the same result as the four semantics for program 5. For program 8, PostgreSQL triggers, the Writes tuples were deleted using the trigger version of rule 2 and then the Publication tuples were deleted using the trigger version of rule 4. For the MySQL implementation, the results depended on the order in which the triggers were written. When the Author triggers were written before the Writes triggers, the tuples with this relation were deleted, and then their associated Publication tuples. When the order was reversed, the Writes tuples were deleted and then their associated Publication tuples. If we would have applied stage semantics instead, only the Author and Writes tuples would have been deleted. Using step semantics, we would have only deleted an Author tuple and the Publication tuples associated with it (regardless of the name of the trigger or the order in which it was written). For program 20, the same number of tuples were deleted by the PostgreSQL triggers as for the four semantics (shown in Figure 6c). The MySQL triggers were not able to terminate computation before the connection to the server was lost. Computing the trigger results for programs 3, 4, and 8 was negligible in terms of execution time for both PostgreSQL and MySQL implementations. For program 20, it was 3.3 minutes for PostgreSQL triggers as opposed to

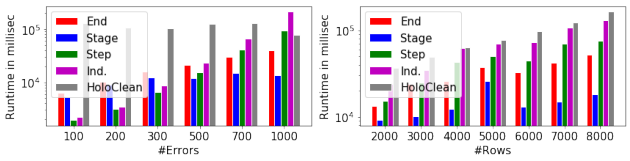
Table 4: Number of over deletions (+) for each of the four semantics compared with number of under repaired tuples (−) by HoloClean for an increasing the number of errors. Note that in contrast to HoloClean all of our semantics always fixed all violations

	Deleted Tuples				Repaired Tuples
Errors	**Ind**	**Step**	**Stage**	**End**	HoloClean
100	**+0**	**+0**	**+389**	**+389**	-26
200	**+0**	**+1**	**+479**	**+479**	-60
300	**+0**	**+5**	**+630**	**+630**	-128
500	**+0**	**+16**	**+786**	**+786**	-234
700	**+0**	**+21**	**+878**	**+878**	-480
1000	**+0**	**+34**	**+1000**	**+1000**	-693

Table 5: Number of tuples that violate a DC with other tuples in the table after/before the repair for both HoloClean and our four semantics. Some tuples participate in multiple violations

	HoloClean					**Semantics**
Errors	DC_1	DC_2	DC_3	DC_4	Total	**Total**
100	22/42	30/46	0/112	0/415	52/615	**0/615**
200	42/82	78/110	0/208	0/563	120/963	**0/963**
300	94/158	98/140	64/302	187/761	443/1361	**0/1361**
500	134/254	116/246	218/500	464/1015	932/2015	**0/2015**
700	198/320	182/364	580/716	872/1272	1832/2672	**0/2672**
1000	238/474	186/520	962/1006	1355/1612	2741/3612	**0/3612**

2.9 minutes for end semantics, and 4.25 minutes for stage semantics, 40.3 minutes for step semantics, and 2.4 minutes for the independent semantics.



(a) Increasing #errors **(b) Increasing #rows**

Figure 10: Runtime comparison with HoloClean for increasing number of errors (rows set to 5000) and number of rows (errors set to 700)

Comparison with HoloClean: HoloClean [44] is a data repair system that relaxes hard constraints (as opposed to our system that views the delta rules as hard constraints) and uses a probabilistic model to infer cell repairs (instead of tuple deletions) in order to clean the database. It leverages DCs, among other methods, to detect and repair cells. HoloClean uses the context of the cell and statistical correlations to repair cells, rather than delete tuples solely based on constraints, as we do for our semantics. In addition, HoloClean does not support cascade deletion. Nevertheless, we have examined what would happen if HoloClean was used in the same context as our system and what would be the difference in results, while also examining the performance of our algorithms for the different semantics in this scenario. We have used the code of the system from [1] with the default configuration that allows for a single table to be inputted. Our comparison used the Author table as presented at the start of this section with an extra attribute stating the organization name: Author(aid, name, oid, organization). We have used four DCs, expressed here as delta rules:

(DC_1) $\Delta_{A_1}(a_1, n_1, o_1, on_1)$:- $A_1(a_1, n_1, o_1, on_1)$, $A_2(a_2, n_2, o_2, on_2)$, $a_1 = a_2, o_1 \neq o_2$

(DC_2) $\Delta_{A_1}(a_1, n_1, o_1, on_1)$:- $A_1(a_1, n_1, o_1, on_1)$, $A_2(a_2, n_2, o_2, on_2)$, $a_1 = a_2, n_1 \neq n_2$

(DC_3) $\Delta_{A_1}(a_1, n_1, o_1, on_1)$:- $A_1(a_1, n_1, o_1, on_1)$, $A_2(a_2, n_2, o_2, on_2)$, $a_1 = a_2, on_1 \neq on_2$

(DC_4) $\Delta_{A_1}(a_1, n_1, o_1, on_1)$:- $A_1(a_1, n_1, o_1, on_1)$, $A_2(a_2, n_2, o_2, on_2)$, $o_1 = o_2, on_1 \neq on_2$

Note that these delta rules simulate DCs semantics. E.g., the first DC says that there cannot be two tuples with the same *aid* and a different *oid* attribute. Thus, if there is such a pair of tuples, the delta rule will delete at least one of them. For these DCs, the results of independent and step semantics should be exact in theory (although our algorithms are heuristic so their output may not be identical to the theoretical results), while the results of end and stage semantics should delete all tuples that satisfy any of these constraints. For Tables 4 and 5 we have taken a table of 5000 rows and increased the number of errors. Table 4 shows the results for the number of tuples deleted beyond the minimum required number by each of our semantics and the difference between the number of repairs to cells made by HoloClean (this is identical to the number of repaired tuples) to the number of required repairs. Algorithm 1 deleted the same number of tuples as the number of errors. The algorithms for the rest of the semantics 'over deleted', while HoloClean has performed fewer repairs than needed[2] outputting an unstable database. In Table 5 we have measured the number of tuples that violate each DC with another tuple after/before the repair for HoloClean, where the "Total" column shows the sum of violations (the sum may be larger than the size of the set, since tuples may participate in violations through multiple DCs). The numbers are the sizes of the results generated by running each DC as an SQL query before and after the repair. As guaranteed by Proposition 3.18 and by our algorithms, every semantics repairs the database so that there are no sets of tuples that violate a DCs, where HoloClean may leave some violating sets of tuples after the repair. Figures 10a and 10b show the runtime performance for all semantics alongside the performance of HoloClean for an increasing number of errors with 5000 rows and for an increasing number of rows with 700 errors. End and stage semantics were faster than the rest, while Algorithms 1 and 2 had similar performance to that of HoloClean.

[2]This is based on the report automatically generated by the system.

7 RELATED WORK

Data repair. Multiple papers have used database constraints as a tool for fixing (in our terms stabilizing) the database [5, 6, 11, 19, 44]. The literature on data repair can be divided by two main criteria: the types of constraints considered and the methods to repair the database. A wide variety of constraints with different forms and functions have been proposed. Examples include functional dependencies and versions thereof [8, 30], and denial constraints [10]. As we have discussed in Section 3.6, our model can express various forms of constraints, but our semantics allow these constraints to be interpreted in different ways and not operate according to one specific algorithm or approach. Regarding methods of data repair, previous works have considered two main approaches: (1) repairing attribute values in cells [6, 11, 29, 33, 44] and (2) tuple deletion [10, 33, 34]; our work focuses on the latter. A major advantage of our approach is the ability to perform cascade deletions over multiple relations in the database while following different well-defined semantics (and the admin may choose which one to follow based on the application scenario). Similar to our independent semantics, a common objective for data repairs is to change the database in the minimal way that will make it consistent with the constraints [5, 19, 33]. In some scenarios a good repair can be obtained by changing values in the database and the metric of minimal changes may not work well [44]. However, in our approach as in [10], we assume that the starting database is *complete*, so the only way to fix it is by deleting tuples and thus we use the minimum cardinality metric to achieve a repair following the delta program; extending delta rules to updates of values is an interesting future work. Similar to our declarative repair framework by delta rules, *declarative data repair* has been explored from multiple angles [20, 21, 28, 43, 51]; e.g., [20] has focused on the rule-based framework of information extraction from text and includes a mechanism for prioritized DC repairs, while [44] expresses constraints in DDlog [48].

Causality in databases. This subject has been the explored in many previous works [36–38]. Works such as [45, 46] consider causal dependencies for explaining results of aggregate queries, that start their operation when there is an initial event of tuple deletion called "intervention" and repair the database if a constraint is violated, e.g., [46] considered repairs with respect to foreign keys in both ways (similar to rules (2) and (3) in Example 1.1). Delta programs can capture these as well as more complex cascaded deletion rules. Moreover, interventions can also be applied in our framework, as we can add auxiliary rules to the program that will start the deletion process.

Stable model. Stable model semantics [14, 24] is a way of defining the semantics of the answer set of logic programs with negation using the concept of a *reduct*. In stable models, if a tuple does not exist in the database, it means that its negation exists. In our model, Δ_i is not the negation of R_i, but is a record of deleted tuples from R_i. Also in our model, the head atom in each rule can only be a delta atom, rather than a positive atom as in stable model. Another relevant work related to our framework is [23], where the authors used the concept of stable models to solve the data conflict problem with trust mappings. The way one's belief is updated from others' beliefs is expressed by weighted update rules that are similar in spirit to our delta rules. However, unlike the delta rules, in [23] rules have priorities, the results of the semantics can be computed in PTime under the *skeptic* paradigm, and they have different usages.

Deletion propagation. Classic deletion propagation is the problem of evaluating the effect of deleting tuples from the database D on the view obtained from evaluating a query Q over D [18, 25, 26]. A more closely related variation is the source side-effect problem [9, 12, 13], which focuses on finding the minimum set of *source tuples in* D that has to be deleted for a given tuple $t \in Q(D)$ to be removed from the result. Our approach may be combined with this problem by including the delta program as another input and solving the source side-effect problem given the delta program and a particular semantics.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a unified framework for data repair that involves deletions. We proposed delta rules to specify the constraints on the data and four semantics that capture behaviors inspired by DCs, a subset of SQL triggers, and causal dependencies, allowing for different interpretation of the same set of constraints. We studied the relationships between these semantics and their complexity, proposed efficient algorithms, and evaluated them experimentally.

We considered delta programs that are not inherently recursive (Section 2). Although all definitions and complexity results (in Sections 2, 3, and 4) apply to recursive programs, our algorithms for intractable cases cannot currently handle recursive programs as for such programs the provenance size may be super-polynomial in the database size. Extending our solutions to recursive programs is a future work. Other future directions include extensions to updates in addition to deletions, and to soft and probabilistic constraints.

Acknowledgements. This research has been funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 804302), the Israeli Science Foundation (ISF) Grant No. 978/17, NSF awards IIS-1552538 and IIS-1703431, NIH award R01EB025021, and the Google Ph.D. Fellowship. The contribution of Amir Gilad is part of his Ph.D. research conducted at Tel Aviv University.

REFERENCES

[1] Code for holoclean. https://github.com/HoloClean/holoclean.

[2] Serge Abiteboul, Meghyn Bienvenu, and Daniel Deutch. Deduction in the presence of distribution and contradictions. In *WebDB*, 2012.

[3] Serge Abiteboul, Daniel Deutch, and Victor Vianu. Deduction with contradictions in datalog. In *ICDT*, pages 143–154, 2014.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] Foto N. Afrati and Phokion G. Kolaitis. Repair checking in inconsistent databases: Algorithms and complexity. In *ICDT*, pages 31–41, 2009.

[6] Leopoldo E. Bertossi, Solmaz Kolahi, and Laks V. S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. *Theory Comput. Syst.*, 52(3):441–482, 2013.

[7] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νz - an optimizing SMT solver. In *TACAS*, pages 194–199, 2015.

[8] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, 2007.

[9] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. On propagation of deletions and annotations through views. In *PODS*, pages 150–158, 2002.

[10] Jan Chomicki and Jerzy Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.*, 197(1-2):90–121, 2005.

[11] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, pages 458–469, 2013.

[12] Gao Cong, Wenfei Fan, and Floris Geerts. Annotation propagation revisited for key preserving views. In *CIKM*, pages 632–641, 2006.

[13] Gao Cong, Wenfei Fan, Floris Geerts, Jianzhong Li, and Jizhou Luo. On the complexity of view update analysis and its application to annotation propagation. *IEEE Trans. Knowl. Data Eng.*, 24(3), 2012.

[14] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[15] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[16] Daniel Deutch and Nave Frost. Constraints-based explanations of classifications. In *ICDE*, pages 530–541, 2019.

[17] Daniel Deutch, Amir Gilad, and Yuval Moskovitch. Selective provenance for datalog programs using top-k queries. *PVLDB*, 8(12):1394–1405, 2015.

[18] Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. Circuits for datalog provenance. In *ICDT*, pages 201–212, 2014.

[19] Ronald Fagin, Benny Kimelfeld, and Phokion G. Kolaitis. Dichotomies in the complexity of preferred repairs. In *PODS*, pages 3–15, 2015.

[20] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Declarative cleaning of inconsistencies in information extraction. *ACM Trans. Database Syst.*, 41(1):6:1–6:44, 2016.

[21] Helena Galhardas, Daniela Florescu, Dennis E. Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB*, pages 371–380, 2001.

[22] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[23] Wolfgang Gatterbauer and Dan Suciu. Data conflict resolution using trust mappings. In *SIGMOD*, pages 219–230, 2010.

[24] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP*, pages 1070–1080, 1988.

[25] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[26] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *VLDB*, pages 675–686, 2007.

[27] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab, United States, 2008.

[28] Jian He, Enzo Veltri, Donatello Santoro, Guoliang Li, Giansalvatore Mecca, Paolo Papotti, and Nan Tang. Interactive and deterministic data cleaning. In *SIGMOD*, pages 893–907, 2016.

[29] Solmaz Kolahi and Laks V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, pages 53–62, 2009.

[30] Nick Koudas, Avishek Saha, Divesh Srivastava, and Suresh Venkatasubramanian. Metric functional dependencies. In *ICDE*, 2009.

[31] Stefan Kratsch, Dániel Marx, and Magnus Wahlström. Parameterized complexity and kernelizability of max ones and exact ones problems. *TOCT*, 8(1):1:1–1:28, 2016.

[32] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic optimization with SMT solvers. In *POPL*, pages 607–618, 2014.

[33] Ester Livshits, Benny Kimelfeld, and Sudeepa Roy. Computing optimal repairs for functional dependencies. In *SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 225–237, 2018.

[34] Andrei Lopatenko and Leopoldo E. Bertossi. Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics. In *ICDT*, pages 179–193, 2007.

[35] MAS. http://academic.research.microsoft.com/.

[36] Alexandra Meliou, Wolfgang Gatterbauer, Joseph Y. Halpern, Christoph Koch, Katherine F. Moore, and Dan Suciu. Causality in databases. *IEEE Data Eng. Bull.*, 33(3):59–67, 2010.

[37] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. WHY so? or WHY no? functional causality for explaining query answers. In *MUD*, pages 3–17, 2010.

[38] Alexandra Meliou, Sudeepa Roy, and Dan Suciu. Causality and explanations in databases. *PVLDB*, 7(13):1715–1716, 2014.

[39] Jim Melton and Alan R Simon. *SQL: 1999: understanding relational language components*. Elsevier, 2001.

[40] MySQL. Mysql trigger syntax. https://dev.mysql.com/doc/refman/8.0/en/trigger-syntax.html, 2019.

[41] Robert Nieuwenhuis and Albert Oliveras. On SAT modulo theories and optimization problems. In *SAT*, pages 156–169, 2006.

[42] PostgreSQL. Postgresql trigger behavior. https://www.postgresql.org/docs/12/trigger-definition.html, 2019.

[43] Nataliya Prokoshyna, Jaroslaw Szlichta, Fei Chiang, Renée J. Miller, and Divesh Srivastava. Combining quantitative and logical data cleaning. *PVLDB*, 9(4):300–311, 2015.

[44] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

[45] Sudeepa Roy, Laurel Orr, and Dan Suciu. Explaining query answers with explanation-ready databases. *PVLDB*, 9(4):348–359, 2015.

[46] Sudeepa Roy and Dan Suciu. A formal approach to finding explanations for database queries. In *SIGMOD*, pages 1579–1590, 2014.

[47] Roberto Sebastiani and Silvia Tomasi. Optimization in SMT with LA(Q) cost functions. *CoRR*, abs/1202.1409, 2012.

[48] Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, and Christopher Ré. Incremental knowledge base construction using deepdive. *PVLDB*, 8(11):1310–1321, 2015.

[49] American National Standard. *Information Systems Database Languages SQL Part 1/2: Framework/Foundation*. American National Standards Institute, Inc., 1999.

[50] TPC. Tpc benchmarks, 2020.

[51] Maksims Volkovs, Fei Chiang, Jaroslaw Szlichta, and Renée J. Miller. Continuous data cleaning. In *ICDE*, pages 244–255, 2014.