Automatic Rollback Suggestions for Incremental Datalog Evaluation

David Zhao[1][0000−0002−3857−5016], Pavle Subotić[2][0000−0002−6536−3932], Mukund Raghothaman[3][0000−0003−2879−0932], and Bernhard Scholz[1][0000−0002−7672−7359]

[1] University of Sydney
{david.zhao,bernhard.scholz}@sydney.edu.au
[2] Microsoft
pavlesubotic@microsoft.com
[3] University Southern California
raghotha@usc.edu

Abstract. Advances in incremental Datalog evaluation strategies have made Datalog popular among use cases with constantly evolving inputs such as static analysis in continuous integration and deployment pipelines. As a result, new logic programming debugging techniques are needed to support these emerging use cases.

This paper introduces an incremental debugging technique for Datalog, which determines the failing changes for a *rollback* in an incremental setup. Our debugging technique leverages a novel incremental provenance method. We have implemented our technique using an incremental version of the Soufflé Datalog engine and evaluated its effectiveness on the DaCapo Java program benchmarks analyzed by the Doop static analysis library. Compared to state-of-the-art techniques, we can localize faults and suggest rollbacks with an overall speedup of over 26.9× while providing higher quality results.

1 Introduction

Datalog has achieved widespread adoption in recent years, particularly in static analysis use cases [8,2,20,19,42,23,4] that can benefit from incremental evaluation. In an industrial setting, static analysis tools are deployed in continuous integration and deployment setups to perform checks and validations after changes are made to a code base [12,1]. Assuming that changes between analysis runs (aka. epochs) are small enough, a static analyzer written in Datalog can be effectively processed by incremental evaluation strategies [40,29,31,28] which recycle computations of previous runs. When a fault appears from a change in the program, users commonly need to (1) localize which changes caused the fault and (2) partially roll back the changes so that the faults no longer appear. However, manually performing this bug localization and the subsequent rollback is impractical, and users typically perform a full rollback while investigating the fault's actual cause [36,37]. The correct change is re-introduced when the fault is

found and addressed, and the program is re-analyzed. This entire debugging process can take significant time. Thus, an automated approach for detecting and performing partial rollbacks can significantly enhance developer productivity.

Existing state-of-the-art Datalog debugging techniques that are available employ data provenance [26,41] or algorithmic debugging [10] to provide explanations. However, these techniques require a deep understanding of the tool's implementation and target the ruleset, not the input. Therefore, such approaches are difficult to apply to automate input localization and rollback. The most natural candidate for this task is *delta debugging* [38,39], a debugging framework for generalizing and simplifying a failing test case. This technique has recently been shown to scale well when integrated with state-of-the-art Datalog synthesizers [30] to obtain better synthesis constraints. Delta debugging uses a divide-and-conquer approach to localize the faults when changes are made to a program, thus providing a concise witness for the fault. However, the standard delta debugging approach is programming language agnostic and requires programs to be re-run, which may require significant time.

In this paper, we introduce a novel approach to automating localize-rollback debugging. Our approach comprises a novel incremental provenance technique and two intertwined algorithms that diagnose and compute a rollback suggestion for a set of faults (missing and unwanted tuples). The first algorithm is a *fault localization* algorithm that reproduces a set of faults, aiding the user in diagnosis. Fault localization traverses the incremental proof tree provided by our provenance technique, producing the subset of an incremental update that causes the faults to appear in the current epoch. The second algorithm performs an *input repair* to provide a local *rollback suggestion* to the user. A rollback suggestion is a subset of an incremental update, such that the faults are fixed when it is rolled back.

We have implemented our technique using an extended incremental version of the Soufflé [24,40] Datalog engine and evaluated its effectiveness on DaCapo [6] Java program benchmarks analyzed by the Doop [8] static analysis tool. Compared to delta debugging, we can localize and fix faults with a speedup of over 26.9× while providing smaller repairs in 27% of the benchmarks. To the best of our knowledge, we are the first to offer such a debugging feature in a Datalog engine, particularly for large workloads within a practical amount of time. We summarize our contributions as follows:

- We propose a novel debugging technique for incremental changing input. We employ localization and rollback techniques for Datalog that scale to real-world program analysis problems.
- We propose a novel incremental provenance mechanism for Datalog engines. Our provenance technique leverages incremental information to construct succinct proof trees.
- We implement our technique in the state-of-the-art Datalog engine Soufflé, including extending incremental evaluation to compute provenance.

– We evaluate our technique with Doop static analysis for large Java programs and compare it to a delta-debugging approach adapted for the localization and rollback problem.

2 Overview

```
admin = new Admin();
sec = new AdminSession();
ins = new InsecureSession();
admin.session = ins;
if (admin.isAdmin && admin.isAuth)
    admin.session = sec;
else
    userSession = ins;
```

(a) Input Program

```
new(admin,L1).
new(sec,L2).
new(ins,L3).
store(admin,session,ins).

store(admin,session,sec).

assign(userSession,ins).
```

(b) EDB Tuples

```
// r1: var = new Obj()
vpt(Var, Obj) :- new(Var, Obj).
// r2: var = var2
vpt(Var, Obj) :- assign(Var, Var2), vpt(Var2, Obj).
// r3: v = i.f; i2.f = v2 where i, i2 point to same obj
vpt(Var, Obj) :- load(Var, Inter, F), store(Inter2, F, Var2),
                 vpt(Inter, InterObj), vpt(Inter2, InterObj),
                 vpt(Var2, Obj).
// r4: v1, v2 point to same obj
alias(V1, V2) :- vpt(V1, Obj), vpt(V2, Obj), V1 != V2.
```

(c) Datalog Points-to Analysis

Fig. 1: Program Analysis Datalog Setup

2.1 Motivating Example

Consider a Datalog pointer analysis in Fig. 1. Here, we show an input program to analyze (Fig. 1a), which is encoded as a set of tuples (Fig. 1b) by an *extractor*, which maintains a mapping between tuples and source code [24,32,34]. We have relations `new`, `assign`, `load`, and `assign` capturing the semantics of the input program to analyze. These relations are also known as the *Extensional Database* (EDB), representing the analyzer's input. The analyzer written in Datalog computes relations `vpt` (*Variable Points To*) and `alias` as the output, which is also known as the *Intensional Database* (IDB). For the points-to analysis, Fig. 1c has four rules. A rule is of the form: $R_h(X_h)$ `:-` $R_1(X_1), \ldots, R_k(X_k)$. Each $R(X)$ is a *predicate*, with R being a *relation* name and X being a vector of *variables* and *constants* of appropriate arity. The predicate to the left of `:-` is the *head* and the sequence of predicates to the right is the *body*. A Datalog rule can be

read from right to left: "for all rule instantiations, if every tuple in the body is derivable, then the corresponding tuple for the head is also derivable".

For example, r_2 is `vpt(Var, Obj) :- assign(Var, Var2), vpt(Var2, Obj)`, which can be interpreted as "if there is an assignment from `Var` to `Var2`, and if `Var2` may point to `Obj`, then also `Var` may point to `Obj`". In combination, the four rules represent a *flow-insensitive* but *field-sensitive* points-to analysis. The IDB relations `vpt` and `alias` represent the analysis result: variables may point to objects and pairs of variables that may be an alias with each other.

Suppose the input program in Fig. 1a changes by adding a method to upgrade a user session to an admin session with the code:

```
upgradedSession = userSession;
userSession = admin.session;
```

The result of the points-to analysis can be incrementally updated by inserting the tuples `assign(upgradedSession, userSession)` and `load(userSession, admin, session)`. After computing the incremental update, we would observe that `alias(userSession, sec)` is now contained in the output. However, we may wish to maintain that `userSession` *should not* alias with the secure session `sec`. Consequently, the incremental update has introduced a *fault*, which we wish to localize and initiate a rollback.

A fault localization provides a subset of the incremental update that is sufficient to reproduce the fault, while a rollback suggestion is a subset of the update which fixes the faults. In this particular situation, the fault localization and rollback suggestion are identical, containing only the insertion of the second tuple, `load(userSession, admin, session)`. Notably, the other tuple in the update, `assign(upgradedSession, userSession)`, is irrelevant for reproducing or fixing the fault and thus is not included in the fault localization/rollback.

In general, an incremental update may contain thousands of inserted and deleted tuples, and a set of faults may contain multiple tuples that are changed in the incremental update. Moreover, the fault tuples may have multiple alternative derivations, meaning that the localization and rollback results are different. In these situations, automatically localizing and rolling back the faults to find a small relevant subset of the incremental update is essential to provide a concise explanation of the faults to the user.

The scenario presented above is common during software development, where making changes to a program causes faults to appear. While our example concerns a points-to analysis computed for a source program, our fault localization and repair techniques are, in principle, applicable to any Datalog program.

Problem Statement: Given an incremental update with its resulting faults, automatically find a fault localization and rollback suggestion.

2.2 Approach Overview

An overview of our approach is shown in Figure 2. The first portion of the system is the incremental Datalog evaluation. Here, the incremental evaluation takes an

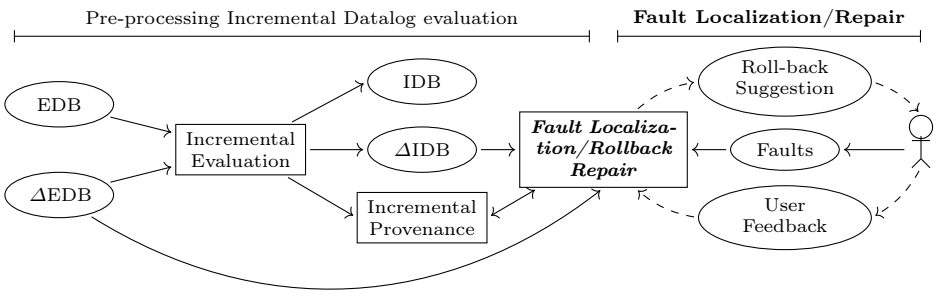

Fig. 2: Fault Localization and Repair System

EDB and an incremental update containing tuples inserted or deleted from the EDB, denoted ΔEDB. The result of the incremental evaluation is the output IDB, along with the set of IDB insertions and deletions from the incremental update, denoted ΔIDB. The evaluation also enables incremental provenance, producing a proof tree for a given query tuple.

The second portion of the system is the fault localization/rollback repair. This process takes a set of faults provided by the user, which is a subset of ΔIDB where each tuple is either unwanted and inserted in ΔIDB or is desirable but deleted in ΔIDB. Then, the fault localization and rollback repair algorithms use the full ΔIDB and ΔEDB, along with incremental provenance, to produce a localization or rollback suggestion.

The main fault localization and rollback algorithms work in tandem to provide localizations or rollback suggestions to the user. The key idea of these algorithms is to compute proof trees for fault tuples using the provenance utility provided by the incremental Datalog engine. These proof trees directly provide localization for the faults. For fault rollback, the algorithms create an Integer Linear Programming (ILP) instance that encodes the proof trees, with the goal of *disabling* all proof trees to prevent the fault tuples from appearing.

The result is a localization or rollback suggestion, which is a subset of ΔEDB. For localization, the subset $S \subseteq \Delta$EDB is such that if we were to apply S to EDB as the diff, the set of faults would be reproduced. For a rollback suggestion, the subset $S \subseteq \Delta$EDB is such that if we were to remove S from ΔEDB, then the resulting diff would *not* produce the faults.

3 Incremental Provenance

Provenance [10,41,30] provides machinery to explain the existence of a tuple. For example, the tuple `vpt(userSession, L3)` could be explained in our running example by the following proof tree:

$$\dfrac{\texttt{assign}(\texttt{userSession}, \texttt{ins}) \qquad \dfrac{\texttt{new}(\texttt{ins}, \texttt{L3})}{\texttt{vpt}(\texttt{ins}, \texttt{L3})}\, r_1}{\texttt{vpt}(\texttt{userSession}, \texttt{L3})}\, r_2$$

However, exploring the proof tree requires familiarity with the Datalog program itself. Thus, provenance alone is an excellent utility for the tool developer but unsuitable for end-users unfamiliar with the Datalog rules.

For fault localization and rollback, a novel provenance strategy is required that builds on incremental evaluation. *Incremental provenance* restricts the computations of the proof tree to the portions affected by the incremental update only. For example, Figure 3 shows an incremental proof tree for the inserted tuple `alias(userSession,sec)`. The tuples labeled with `(+)` are inserted by an incremental update. Incremental provenance would only compute provenance information for these newly inserted tuples and would not explore the tuples in red already established in a previous epoch.

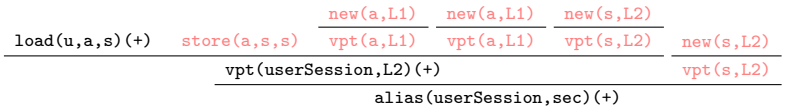

Fig. 3: The proof tree for `alias(userSession,sec)`. `(+)` denotes tuples that are inserted as a result of the incremental update, red denotes tuples that were not affected by the incremental update.

To formalize incremental provenance, we define inc-prov as follows. Given an incremental update ΔE, inc-prov$(t, \Delta E)$ should consist of tuples that were updated due to the incremental update.

Definition 1. *The set* inc-prov$(t, \Delta E)$ *is the set of tuples that appear in the proof tree for* t*, that are also inserted as a result of* ΔE*.*

A two-phase approach for provenance was introduced in [41]. In the first phase, tuples are annotated with *provenance annotations* while computing the IDB. In the second phase, the user can query a tuple's proof tree using the annotations from the first phase. For each tuple, the important annotation is its minimal proof tree height. For instance, our running example produces the tuple `vpt(userSession, L3)`. This tuple would be annotated with its minimal proof tree height of 2. Formally, the height annotation is 0 for an EDB tuple, or $h(t) = \max\{h(t_1), \ldots, h(t_k)\} + 1$ if t is computed by a rule instantiation t :- $t_1, \ldots, t_k$. The provenance annotations are used in a provenance construction stage, where the annotations form constraints to guide the proof tree construction.

For incremental evaluation, the standard strategies [29,28,31,40] use *incremental annotations* to keep track of when tuples are computed during the execution. In particular, for each tuple, [40] stores the iteration number of the fixpoint computation and a count for the number of derivations. To compute provenance information in an incremental evaluation setting, we observe a correspondence between the iteration number and provenance annotations. A tuple is produced in some iteration if at least one of the body tuples was produced in the previous iteration. Therefore, the iteration number I for a tuple produced in a fixpoint is

equivalent to $I(t) = \max\{I(t_1), \ldots, I(t_k)\} + 1$ if t is computed by rule instantiation t :- $t_1, \ldots, t_k$. This definition of iteration number corresponds closely to the height annotation in provenance. Therefore, the iteration number is suitable for constructing proof trees similar to provenance annotations.

For fault localization and rollback, it is also important that the Datalog engine produces only provenance information that is *relevant* for faults that appear after an incremental update. Therefore, the provenance information produced by the Datalog engine should be restricted to tuples inserted or deleted by the incremental update. Thus, we adapt the user-driven proof tree exploration process in [41] to use an automated procedure that enumerates exactly the portions of the proof tree that have been affected by the incremental update.

As a result, our approach for incremental provenance produces proof trees containing only tuples inserted or deleted due to an update. For fault localization and rollback, this property is crucial for minimizing the search space when computing localizations and rollback suggestions.

4 Fault Localization and Rollback Repair

This section describes our approach and algorithms for both the fault localization and rollback problems. We begin by formalizing the problem and then presenting basic versions of both problems. Finally, we extend the algorithms to handle missing faults and negation.

4.1 Preliminaries

We first define a fault to formalize the fault localization and rollback problems. For a Datalog program, a fault may manifest as either (1) an undesirable tuple that appears or (2) a desirable tuple that disappears. In other words, a fault is a tuple that does not match the *intended semantics* of a program.

Definition 2 (Intended Semantics). *The* intended semantics *of a Datalog program P is a pair of sets (I_+, I_-) where I_+ and I_- are desirable and undesirable tuple sets, respectively. An input set E is* correct *w.r.t P and (I_+, I_-) if $I_+ \subseteq P(E)$ and $I_- \cap P(E) = \emptyset$.*

Given an intended semantics for a program, a *fault* can be defined as follows:

Definition 3 (Fault). *Let P be a Datalog program, with input set E and intended semantics (I_+, I_-). Assume that E is incorrect w.r.t. P with (I_+, I_-). Then, a* fault *of E is a tuple t such that either t is desirable but missing, i.e., $t \in I_+ \setminus P(E)$ or t is undesirable but produced, i.e., $t \in P(E) \cap I_-$.*

We can formalize the situation where an incremental update for a Datalog program introduces a fault. Let P be a Datalog program with intended semantics $I_\checkmark = (I_+, I_-)$ and let E_1 be an input EDB. Then, let $\Delta E_{1\to 2}$ be an incremental update, such that $E_1 \uplus \Delta E_{1\to 2}$ results in another input EDB, E_2. Then, assume that E_1 is correct w.r.t $I_\checkmark$, but E_2 is incorrect.

Fault Localization. The fault localization problem allows the user to pinpoint the sources of faults. This is achieved by providing a minimal subset of the incremental update that can still reproduce the fault.

Definition 4 (Fault Localization). *A* fault localization *is a subset* $\delta E \subseteq \Delta E_{1\to 2}$ *such that* $P(E_1 \uplus \delta E)$ *exhibits* all *faults of* E_2.

Rollback Suggestion. A rollback suggestion provides a subset of the diff, such that its removal from the diff would fix all faults.

Definition 5 (Rollback Suggestion). *A* rollback suggestion *is a subset* $\delta E_\times \subseteq \Delta E_{1\to 2}$ *such that* $P(E_1 \uplus (\Delta E_{1\to 2} \setminus \delta E_\times))$ *does not produce any faults of* E_2.

Ideally, fault localizations and rollback suggestions should be of minimal size.

4.2 Fault Localization

In the context of incremental Datalog, the *fault localization problem* provides a small subset of the incremental changes that allow the fault to be reproduced.

We begin by considering a basic version of the fault localization problem. In this basic version, we have a positive Datalog program (i.e., with no negation), and we localize a set of faults that are undesirable but appear (i.e., $P(E) \cap I_-$). The main idea of the fault localization algorithm is to compute a proof tree for each fault tuple. The tuples forming these proof trees are sufficient to localize the faults since these tuples allow the proof trees to be valid and, thus, the fault tuples to be reproduced.

Algorithm 1 Localize-Faults(P, E_2, $\Delta E_{1\to 2}$, F): Given a diff $\Delta E_{1\to 2}$ and a set of fault tuples F, returns $\delta E \subseteq \Delta E_{1\to 2}$ such that $E_1 \uplus \delta E$ produces all $t \in F$

1: **for** tuple $t \in F$ **do**
2: Let inc-prov$(t, \Delta E_{1\to 2})$ be an incremental proof tree of t w.r.t P and E_2, containing tuples that were inserted due to $\Delta E_{1\to 2}$
3: **return** $\cup_{t\in F}$(inc-prov$(t, \Delta E_{1\to 2}) \cap \Delta E_{1\to 2}$)

The basic fault localization is presented in Alg. 1. For each fault tuple $t \in F$, the algorithm computes one incremental proof tree inc-prov$(t, \Delta E_{1\to 2})$. These proof trees contain the set of tuples that were inserted due to the incremental update $\Delta E_{1\to 2}$ and cause the existence of each fault tuple t. Therefore, by returning the union $\cup_{t\in F}$(inc-prov$(t, \Delta E_{1\to 2}) \cap \Delta E_{1\to 2}$), the algorithm produces a subset of $\Delta E_{1\to 2}$ that reproduces the faults.

4.3 Rollback Repair

The rollback repair algorithm produces a rollback suggestion. As with fault localization, we begin with a basic version of the rollback problem, where we have

only a positive Datalog program and wish to roll back a set of unwanted fault tuples. The basic rollback repair algorithm involves computing *all* non-cyclic proof trees for each fault tuple and 'disabling' each of those proof trees, as shown in Alg. 2. If all proof trees are invalid, the fault tuple will no longer be computed by the resulting EDB.

Algorithm 2 Rollback-Repair(P, E_2, $\Delta E_{1\to 2}$, F): Given a diff $\Delta E_{1\to 2}$ and a set of fault tuples F, return a subset $\delta E \subseteq \Delta E_{1\to 2}$ such that $E_1 \uplus (\Delta E_{1\to 2} \setminus \delta E)$ does not produce t_r

1: Let all-inc-prov$(t, \Delta E_{1\to 2}) = \{T_1, \ldots, T_n\}$ be the total incremental provenance for a tuple t w.r.t P and E_2, where each T_i is a non-cyclic proof tree containing tuples inserted due to $\Delta E_{1\to 2}$.
Construct an integer linear program instance as follows:
2: Create a 0/1 integer variable x_{t_k} for each tuple t_k that occurs in the proof trees in all-inc-prov$(t, \Delta E_{1\to 2})$ for each fault tuple $t \in F$
3: **for** each tuple $t_f \in F$ **do**
4: **for** each proof tree $T_i \in$ all-inc-prov$(t, \Delta E_{1\to 2})$ **do**
5: **for** each line $t_h \leftarrow t_1 \wedge \ldots \wedge t_k$ in T_i **do**
6: Add a constraint $x_{t_1} + \ldots + x_{t_k} - x_{t_h} \leq k - 1$
7: Add a constraint $x_{t_f} = 0$
8: Add the objective function maximize $\sum_{t_e \in EDB} x_{t_e}$
9: Solve the ILP
10: Return $\{t \in \Delta E_{1\to 2} \mid x_t = 0\}$

Alg. 2 computes a minimum subset of the diff $\Delta E_{1\to 2}$, which would prevent the production of each $t \in F$ when excluded from the diff. The key idea is to use integer linear programming (ILP) [33] as a vehicle to disable EDB tuples so that the fault tuples vanish in the IDB. We phrase the proof trees as a pseudo-Boolean formula [22] whose propositions represent the EDB and IDB tuples. In the ILP, the faulty tuples are constrained to be false, and the EDB tuples assuming the true value are to be maximized, i.e., we wish to eliminate the least number of tuples in the EDB for repair. The ILP created in Alg. 2 introduces a variable for each tuple (either IDB or EDB) that appears in *all* incremental proof trees for the fault tuples. For the ILP model, we have three types of constraints: (1) to encode a single-step proof, (2) to enforce that fault tuples are false, and (3) to ensure that variables are in the 0-1 domain. The constraints encoding proof trees correspond to each one-step derivation which can be expressed as a Boolean constraint $t_1 \wedge \ldots \wedge t_k \implies t_h$ for the rule application, where $t_1, \ldots, t_k$ and t_h are Boolean variables. Using propositional logic rules, this is equivalent to $\overline{t_1} \vee \ldots \vee \overline{t_k} \vee t_h$. This formula is then transformed into a pseudo (linear) Boolean formula where φ maps a Boolean function to the $0-1$ domain, and x_t

corresponds to the 0-1 variable of proposition t in the ILP.

$$\begin{aligned}\varphi\left(\overline{t_1} \vee \ldots \vee \overline{t_k} \vee t_h\right) &\equiv (1 - x_{t_1}) + \ldots + (1 - x_{t_k}) + t_h > 0 \\ &\equiv x_{t_1} + \ldots + x_{t_k} - x_{t_h} \leq k - 1\end{aligned}$$

The constraints assuming false values for fault tuples $t_f \in F$ are simple equalities, i.e., $x_{t_f} = 0$. The objective function for the ILP is to maximize the number of inserted tuples that are kept, which is equivalent to minimizing the number of tuples in $\Delta E_{1\to 2}$ that are disabled by the repair. In ILP form, this is expressed as maximizing $\sum_{t\in\Delta E_{1\to 2}} t$.

$$\begin{array}{lll}\text{max.} & \sum_{t\in\Delta E_{1\to 2}} x_t & \\ \text{s.t.} & x_{t_1} + \ldots x_{t_k} - x_{t_h} \leq k - 1 & (\forall t_h \Leftarrow t_1 \wedge \ldots \wedge t_k \in T_i) \\ & x_{t_f} = 0 & (\forall t_f \in F) \\ & x_t \in \{0, 1\} & (\forall \text{tuples } t)\end{array}$$

The solution of the ILP permits us to determine the EDB tuples for repair:

$$\delta E = \{t \in \Delta E_{1\to 2} \mid x_t = 0\}$$

This is a minimal set of inserted tuples that must be removed from $\Delta E_{1\to 2}$ so that the fault tuples disappear.

This ILP formulation encodes the problem of disabling all proof trees for all fault tuples while maximizing the number of inserted tuples kept in the result. If there are multiple fault tuples, the algorithm computes proof trees for each fault tuple and combines all proof trees in the ILP encoding. The result is a set of tuples that is minimal but sufficient to prevent the fault tuples from being produced.

4.4 Extensions

Missing Tuples. The basic versions of the fault localization and rollback repair problem only handle a tuple which is undesirable but appears. The opposite kind of fault, i.e., a tuple which is desirable but missing, can be localized or repaired by considering a *dual* version of the problem. For example, consider a tuple t that disappears after applying a diff $\Delta E_{1\to 2}$, and appears in the update in the opposite direction, $\Delta E_{2\to 1}$. Then, the dual problem of localizing the disappearance of t is to *rollback* the appearance of t after applying the opposite diff, $\Delta E_{2\to 1}$.

To localize a disappearing tuple t, we want to provide a small subset δE of $\Delta E_{1\to 2}$ such that t is still not computable after applying δE to E_1. To achieve this, *all* ways to derive t must be invalid after applying δE. Considering the dual problem, rolling back the appearance of t in $\Delta E_{2\to 1}$ results in a subset δE such that $E_2 \uplus (\Delta E_{2\to 1} \setminus \delta E)$ does not produce t. Since $E_1 = E_2 \uplus \Delta E_{2\to 1}$, if we were to apply the reverse of δE (i.e., insertions become deletions and vice versa), we would arrive at the same EDB set as $E_2 \uplus (\Delta E_{2\to 1} \setminus \delta E)$. Therefore, the reverse of δE is the desired minimal subset that localizes the disappearance of t.

Similarly, to roll back a disappearing tuple t, we apply the dual problem of *localizing* the appearance of t after applying the opposite diff $\Delta E_{2\to1}$. Here, to roll back a disappearing tuple, we introduce *one* way of deriving t. Therefore, localizing the appearance of t in the opposite diff provides one derivation for t and thus is the desired solution. In summary, to localize or rollback a tuple t that is missing after applying $\Delta E_{1\to2}$, we compute a solution for the dual problem. The dual problem for localization is to roll back the appearance of t after applying $\Delta E_{2\to1}$, and similarly, the dual problem for rollback is localization.

Stratified Negation. Stratified negation is a common extension for Datalog. With stratified negation, atoms in the body of a Datalog rule may appear negated, e.g., $R_h(X_h)$ `:-` $R_1(X_1), \ldots, !R_k(X_k), \ldots, R_n(X_n)$. The negated atoms are denoted with !, and any variables contained in negated atoms must also exist in some positive atom in the body of the rule (a property called *groundedness*). Semantically, negations are satisfied if the instantiated tuple *does not* exist in the corresponding relation. The 'stratified' in 'stratified negation' refers to the property that no cyclic negations can exist. For example, the rule $A(x)$ `:-` $B(x, y), !A(y)$ causes a dependency cycle where A depends on the negation of A and thus is not allowed under stratified negation.

Consider the problem of localizing the appearance of an unwanted tuple t. If the Datalog program contains stratified negation, then the appearance of t can be caused by two possible situations. Either (1) there is a positive tuple in the proof tree of t that appears, or (2) there is a negated tuple in the proof tree of t that disappears. The first case is the standard case, but in the second case, if a negated tuple disappears, then its disappearance can be localized or rolled back by computing the dual problem as in the missing tuple strategy presented above. We may encounter further negated tuples in executing the dual version of the problem. For example, consider the set of Datalog rules $A(x)$ `:-` $B(x), !C(x)$ and $C(x)$ `:-` $D(x), !E(x)$. If we wish to localize an appearing (unwanted) tuple `A(x)`, we may encounter a disappearing tuple `C(x)`. Then, executing the dual problem, we may encounter an appearing tuple `E(x)`. We can generally continue flipping between the dual problems to solve the localization or repair problem. This process is guaranteed to terminate due to the stratification of negations. Each time the algorithm encounters a negated tuple, it must appear in an earlier stratum than the previous negation. Therefore, eventually, the negations will reach the input EDB, and the process terminates.

4.5 Full Algorithm

The full rollback repair algorithm presented in Alg. 3 incorporates the basic version of the problem and all of the extensions presented above. The result of the algorithm is a rollback suggestion, which fixes all faults. Alg. 3 begins by initializing the EDB after applying the diff (line 1) and separate sets of unwanted faults (lines 2) and missing faults (3). The set of candidate tuples forming the repair is initialized to be empty (line 4).

Algorithm 3 Full-Rollback-Repair(P, E_1, $\Delta E_{1\to 2}$, (I_+, I_-)): Given a diff $\Delta E_{1\to 2}$ and an intended semantics $(I_+, I_,)$, compute a subset $\delta E \subseteq \Delta E_{1\to 2}$ such that $\Delta E_{1\to 2} \setminus \delta E$ satisfies the intended semantics

```
1: Let E_2 be the EDB after applying the diff: E_1 ⊎ ΔE_{1→2}
2: Let F^+ be appearing unwanted faults: {I_- ∩ P(E_2)}
3: Let F^- be missing desirable faults: {I_+ \ P(E_2)}
4: Let L be the set of repair tuples, initialized to ∅
5: while both F^+ and F^- are non-empty do
6:     Add Rollback-Repair(P, E_2, ΔE_{1→2}, F^+) to L
7:     for negated tuples !t ∈ L do
8:         Add t to F^-
9:     Clear F^+
10:    Add Localize-Faults(P, E_1, ΔE_{2→1}, F^-) to L
11:    for negated tuples !t ∈ L do
12:        Add t to F^+
13:    Clear F^-
14: return L
```

The main part of the algorithm is a worklist loop (lines 5 to 13). In this loop, the algorithm first processes all unwanted but appearing faults (F^+, line 6) by computing the repair of F^+. The result is a subset of tuples in the diff such that the faults F^+ no longer appear when the subset is excluded from the diff. In the provenance system, negations are treated as EDB tuples, and thus the resulting repair may contain negated tuples. These negated tuples are added to F^- (line 7) since a tuple appearing in F^+ may be caused by a negated tuple disappearing. The algorithm then repairs the tuples in F^- by computing the dual problem, i.e., localizing F^- with respect to $\Delta E_{2\to 1}$. Again, this process may result in negated tuples, which are added to F^+, and the loop begins again. This worklist loop must terminate, due to the semantics of stratified negation, as discussed above. At the end of the worklist loop, L contains a candidate repair.

While Alg. 3 presents a full algorithm for rollback, the fault localization problem can be solved similarly. Since rollback and localization are dual problems, the full fault localization algorithm swaps Rollback-Repair in line 6 and Localize-Faults in line 10.

Example We demonstrate how our algorithms work by using our running example. Recall that we introduce an incremental update consisting of inserting two tuples `assign(upgradedSession, userSession)` and `load(userSession, admin, session)`. As a result, the system computes the unwanted fault tuple `alias(userSession, sec)`. To rollback the appearance of the fault tuple, the algorithms start by computing its provenance, as shown in Figure 3. The algorithm then creates a set of ILP constraints, where each tuple (with shortened

variables) represents an ILP variable:

$$\text{maximize} \sum \texttt{load}(\texttt{u}, \texttt{a}, \texttt{s}) \text{ such that}$$
$$\texttt{load}(\texttt{u}, \texttt{a}, \texttt{s}) - \texttt{vpt}(\texttt{u}, \texttt{L2}) \leq 0,\ \texttt{vpt}(\texttt{u}, \texttt{L2}) - \texttt{alias}(\texttt{u}, \texttt{s}) \leq 0,\ \texttt{alias}(\texttt{u}, \texttt{s}) = 0$$

For this simple ILP, the result indicates that the insertion of `load(userSession, admin, session)` should be rolled back to fix the fault.

5 Experiments

This section evaluates our technique on real-world benchmarks to determine its effectiveness and applicability. We consider the following research questions:

- RQ1: Is the new technique faster than a delta-debugging strategy?
- RQ2: Does the new technique produce more precise localization/repair candidates than delta debugging?

Experimental Setup: [4] We implemented the fault localization and repair algorithms using Python[5]. The Python code calls out to an incremental version of the Soufflé Datalog engine [40] extended with incremental provenance. Our implementation of incremental provenance uses the default metric of minimizing proof tree height, as it provides near-optimal repairs with slight runtime improvements. For solving integer linear programs, we use the GLPK library.

Our main point of comparison in our experimental evaluation is the delta debugging approach, such as that used in the ProSynth Datalog synthesis framework [30]. We adapted the implementation of delta debugging used in ProSynth to support input tuple updates. Like our fault repair implementation, the delta debugging algorithm was implemented in Python; however, it calls out to the standard Soufflé engine since that provides a lower overhead than the incremental or provenance versions.

For our benchmarks, we use the Doop program analysis framework [8] with the DaCapo set of Java benchmarks [7]. The analysis contains approx. 300 relations, 850 rules, and generates approx. 25 million tuples from an input size of 4-9 million tuples per DaCapo benchmark. For each of the DaCapo benchmarks, we selected an incremental update containing 50 tuples to insert and 50 tuples to delete, which is representative of the size of a typical commit in the underlying source code. From the resulting IDB changes, we selected four different arbitrary fault sets for each benchmark, which may represent an analysis error.

Performance: The results of our experiments are shown in Table 1. Our fault repair technique performs much better overall compared to the delta debugging technique. We observe a geometric mean speedup of over 26.9× [6] compared to

[4] We use an Intel Xeon Gold 6130 with 192 GB RAM, GCC 10.3.1, and Python 3.8.10
[5] Available at `github.com/davidwzhao/souffle-fault-localization`
[6] We say "over" because we bound timeouts to 7200 seconds.

Table 1: Repair size and runtime of our technique compared to delta debugging

		Rollback Repair				Delta Debugging		
Program	No.	Size	Overall (s)	Local(s)	Repair(s)	Size	Runtime (s)	Speedup
antlr	1	2	73.6	0.51	73.1	3	3057.8	41.5
	2	1	79.4	0.00	79.4	1	596.5	7.5
	3	1	0.95	0.95	-	1	530.8	558.7
	4	2	77.8	1.89	75.9	3	3017.6	38.8
bloat	1	2	3309.5	0.02	3294.1	2	2858.6	0.9
	2	1	356.3	0.00	355.4	1	513.6	1.4
	3	1	0.33	0.33	-	1	557.7	1690.0
	4	3	3870.6	0.10	3854.7	2	2808.3	0.7
chart	1	1	192.6	0.00	192.6	1	685.0	3.6
	2	1	3.01	3.01	-	1	675.3	224.4
	3	1	78.8	0.00	78.8	1	667.6	8.5
	4	2	79.9	3.24	76.7	3	3001.1	37.6
eclipse	1	2	177.3	0.04	177.2	3	2591.2	14.6
	2	1	79.2	0.00	79.1	1	416.1	5.3
	3	1	0.12	0.12	-	1	506.3	4219.2
	4	3	91.9	0.09	91.8	3	2424.4	26.4
fop	1	2	83.8	0.05	83.8	2	3446.6	41.1
	2	1	76.9	0.00	76.9	1	670.7	8.7
	3	1	0.66	0.66	-	1	721.8	1093.6
	4	6	74.8	0.50	74.3	Timeout (7200)		96.3+
hsqldb	1	2	83.3	0.04	83.3	2	2979.2	35.8
	2	1	79.4	0.00	79.4	1	433.8	5.5
	3	1	74.0	0.00	74.0	1	663.1	9.0
	4	3	75.5	0.04	75.5	5	6134.8	81.3
jython	1	1	83.3	0.00	83.3	1	609.4	7.3
	2	1	78.2	0.00	78.2	1	590.4	7.5
	3	1	76.6	0.00	76.6	1	596.2	7.8
	4	1	75.8	0.00	75.8	1	587.6	7.8
luindex	1	2	81.3	0.07	81.2	3	2392.1	29.4
	2	1	79.8	0.00	79.8	1	511.0	6.4
	3	1	0.10	0.10	-	1	464.8	4648.0
	4	4	77.9	0.12	77.8	5	4570.4	58.7
lusearch	1	2	110.2	0.06	110.0	3	2558.8	23.2
	2	1	1062.1	0.00	1057.4	1	370.4	0.3
	3	1	0.12	0.12	-	1	369.6	3080.0
	4	2	294.2	0.06	293.2	3	2420.9	8.2
pmd	1	2	78.1	0.02	78.1	3	3069.8	39.3
	2	1	77.0	0.00	77.0	1	600.2	7.8
	3	1	0.08	0.08	-	1	717.8	8972.5
	4	3	74.3	0.08	74.2	3	2828.3	38.1
xalan	1	1	84.9	0.00	84.9	1	745.3	8.8
	2	1	82.2	0.00	82.2	1	728.9	8.9
	3	1	100.1	0.00	100.1	1	1243.7	12.4
	4	1	521.6	0.00	518.3	1	712.5	1.4

delta debugging. For delta debugging, the main cause of performance slowdown is that it is a black-box search technique, and it requires multiple iterations of invoking Soufflé (between 6 and 19 invocations for the presented benchmarks) to direct the search. This also means that any intermediate results generated in a previous Soufflé run are discarded since no state is kept to allow the reuse of results. Each invocation of Soufflé takes between 30-50 seconds, depending on the benchmark and EDB. Thus, the overall runtime for delta debugging is in the hundreds of seconds at a minimum. Indeed, we observe that delta debugging takes between 370 and 6135 seconds on our benchmarks, with one instance timing out after two hours (7200 seconds).

On the other hand, our rollback repair technique calls for provenance information from an already initialized instance of incremental Soufflé. This incrementality allows our technique to reuse the already computed IDB for each provenance query. For eight of the benchmarks, the faults only contained missing tuples. Therefore, only the Localize-Faults method was called, which only computes one proof tree for each fault tuple and does not require any ILP solving. The remainder of the benchmarks called the Rollback-Repair method, where the main bottleneck is for constructing and solving the ILP instance. For three of the benchmarks, `bloat-1`, `bloat-4`, and `lusearch-2`, the runtime was slower than delta debugging. This result is due to the fault tuples in these benchmarks having many different proof trees, which took longer to compute. In addition, this multitude of proof trees causes a larger ILP instance to be constructed, which took longer to solve.

Quality: While the delta debugging technique produces 1-minimal results, we observe that despite no overall optimality guarantees, our approach was able to produce more minimal repairs in 27% of the benchmarks. Moreover, our rollback repair technique produced a larger repair in only one of the benchmarks. This difference in quality is due to the choices made during delta debugging. Since delta debugging has no view of the internals of Datalog execution, it can only partition the EDB tuples randomly. Then, the choices made by delta debugging may lead to a locally minimal result that is not globally optimal. For our fault localization technique, most of the benchmarks computed one iteration of repair and did not encounter any negations. Therefore, due to the ILP formulation, the results were optimal in these situations. Despite our technique overall not necessarily being optimal, it still produces high-quality results in practice.

6 Related Work

Logic Programming Input Repair. A plethora of logic programming paradigms exist that can express diagnosis and repair by EDB regeneration [25,16,18,13]. Unlike these logic programming paradigms, our technique is designed to be embedded in high-performance modern Datalog engines. Moreover, our approach can previous computations (proof trees and incremental updates) to localize and repair only needed tuples. This bounds the set of repair candidates and results in

apparent speedups. Other approaches, such as the ABC Repair System [27], use a combination of provenance-like structures and user-guided search to localize and repair faults. However, that approach is targeted at the level of the Datalog specification and does not always produce effective repairs. Techniques such as delta debugging have recently been used to perform state-of-the-art synthesis of Datalog programs efficiently [30]. Our delta debugging implementation adapts this method, given it produces very competitive synthesis performance and can be easily re-targeted to diagnose and repair inputs.

Database Repair. Repairing inconsistent databases with respect to integrity constraints has been extensively investigated in the database community [15,9,3,17]. Unlike our approach, integrity constraints are much less expressive than Datalog; in particular, they do not allow fixpoints in their logic. The technique in [17] shares another similarity in that it also presents repair for incremental SQL evaluation. However, this is limited to relational algebra, i.e., SQL and Constrained Functional Dependencies (CFDs) that are less expressive than Datalog. A more related variant of database repair is consistent query answering (CQA)[9,3]. These techniques repair query answers given a database, integrity constraints and an SQL query. Similarly, these approaches do not support recursive queries, as can be expressed by Datalog rules.

Program Slicing. Program slicing [35,5,14,21] encompasses several techniques that aim to compute portions (or *slices*) of a program that contribute to a particular output result. For fault localization and debugging, program slicing can be used to localize slices of programs that lead to a fault or error. The two main approaches are *static* program slicing, which operates on a static control flow graph, and *dynamic* program slicing, which considers the values of variables or execution flow of a particular execution. As highlighted by [11], data provenance is closely related to slicing. Therefore, our technique can be seen as a form of static slicing of the Datalog EDB with an additional rollback repair stage.

7 Conclusion

We have presented a new debugging technique that localizes faults and provides rollback suggestions for Datalog program inputs. Unlike previous approaches, our technique does not entirely rely on a black-box solver to perform the underlying repair. Instead, we utilize incremental provenance information. As a result, our technique exhibits speedups of 26.9× compared to delta debugging and finds more minimal repairs 27% of the time.

There are also several potential future directions for this research. One is to adapt our technique to repair changes in Datalog *rules*, as well as changes in input tuples. Another direction is to adopt these techniques for different domain areas outside the use cases of program analyses.

Acknowledgments M.R. was funded by U.S. NSF grants CCF-2146518, CCF-2124431, and CCF-2107261.

References

1. Github codeql (2021), https://codeql.github.com/, accessed: 19-10-2021
2. Allen, N., Scholz, B., Krishnan, P.: Staged Points-to Analysis for Large Code Bases, pp. 131–150. Springer Berlin Heidelberg (2015). https://doi.org/10.1007/978-3-662-46663-6_7
3. Arenas, M., Bertossi, L.E., Chomicki, J.: Answer sets for consistent query answering in inconsistent databases. Theory Pract. Log. Program. **3**(4-5), 393–424 (2003)
4. Backes, J., Bayless, S., Cook, B., Dodge, C., Gacek, A., Hu, A.J., Kahsai, T., Kocik, B., Kotelnikov, E., Kukovec, J., McLaughlin, S., Reed, J., Rungta, N., Sizemore, J., Stalzer, M.A., Srinivasan, P., Subotic, P., Varming, C., Whaley, B.: Reachability analysis for aws-based networks. In: Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. pp. 231–241 (2019)
5. Binkley, D.W., Gallagher, K.B.: Program slicing. Advances in computers **43**, 1–50 (1996)
6. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications. pp. 169–190. ACM Press, New York, NY, USA (Oct 2006). https://doi.org/http://doi.acm.org/10.1145/1167473.1167488
7. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., et al.: The dacapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. pp. 169–190 (2006)
8. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. SIGPLAN Not. **44**(10), 243–262 (2009). https://doi.org/10.1145/1639949.1640108, http://doi.acm.org/10.1145/1639949.1640108
9. Bravo, L., Bertossi, L.E.: Consistent query answering under inclusion dependencies. In: Lutfiyya, H., Singer, J., Stewart, D.A. (eds.) Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, October 5-7, 2004, Markham, Ontario, Canada. pp. 202–216. IBM (2004)
10. Caballero, R., Riesco, A., Silva, J.: A survey of algorithmic debugging. ACM Computing Surveys (CSUR) **50**(4), 60 (2017)
11. Cheney, J.: Program slicing and data provenance. IEEE Data Eng. Bull. **30**(4), 22–28 (2007)
12. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at facebook. Commun. ACM **62**(8), 62–70 (Jul 2019)
13. El-Hassany, A., Tsankov, P., Vanbever, L., Vechev, M.: Network-wide configuration synthesis. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. pp. 261–281. Springer International Publishing, Cham (2017)
14. Ezekiel, S., Lukas, K., Marcel, B., Zeller, A.: Locating faults with program slicing: an empirical analysis. Empirical Software Engineering **26**(3) (2021)
15. Fan, W.: Constraint-Driven Database Repair, pp. 458–463. Springer US, Boston, MA (2009). https://doi.org/10.1007/978-0-387-39940-9_599, https://doi.org/10.1007/978-0-387-39940-9_599

16. Fan, W., Geerts, F., Jia, X.: Semandaq: A data quality system based on conditional functional dependencies. Proc. VLDB Endow. **1**(2), 1460–1463 (aug 2008)
17. Fan, W., Geerts, F., Jia, X.: Semandaq: A data quality system based on conditional functional dependencies. Proc. VLDB Endow. **1**(2), 1460–1463 (aug 2008). https://doi.org/10.14778/1454159.1454200, https://doi.org/10.14778/1454159.1454200
18. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. pp. 1070–1080. MIT Press (1988)
19. Grech, N., Brent, L., Scholz, B., Smaragdakis, Y.: Gigahorse: Thorough, declarative decompilation of smart contracts. In: Proceedings of the 41th International Conference on Software Engineering, ICSE 2019. p. (to appear). ACM, Montreal, QC, Canada (May 2019)
20. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: Surviving out-of-gas conditions in ethereum smart contracts. In: SPLASH 2018 OOPSLA (2018)
21. Harman, M., Hierons, R.: An overview of program slicing. software focus **2**(3), 85–92 (2001)
22. Hooker, J.: Generalized resolution for 0–1 linear inequalities. Annals of Mathematics and Artificial Intelligence **6**, 271–286 (03 1992). https://doi.org/10.1007/BF01531033
23. Huang, S.S., Green, T.J., Loo, B.T.: Datalog and emerging applications: An interactive tutorial. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. pp. 1213–1216. SIGMOD '11, ACM (2011). https://doi.org/10.1145/1989323.1989456, http://doi.acm.org/10.1145/1989323.1989456
24. Jordan, H., Scholz, B., Subotić, P.: Soufflé: On synthesis of program analyzers. In: International Conference on Computer Aided Verification. pp. 422–430. Springer (2016)
25. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive logic programming (1993)
26. Karvounarakis, G., Ives, Z.G., Tannen, V.: Querying data provenance. p. 951–962. SIGMOD '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1807167.1807269, https://doi.org/10.1145/1807167.1807269
27. Li, X., Bundy, A., Smaill, A.: Abc repair system for datalog-like theories. In: KEOD. pp. 333–340 (2018)
28. McSherry, F., Murray, D.G., Isaacs, R., Isard, M.: Differential dataflow. In: CIDR (2013)
29. Motik, B., Nenov, Y., Piro, R., Horrocks, I.: Maintenance of datalog materialisations revisited. Artificial Intelligence **269**, 76–136 (2019)
30. Raghothaman, M., Mendelson, J., Zhao, D., Naik, M., Scholz, B.: Provenance-guided synthesis of datalog programs. Proceedings of the ACM on Programming Languages **4**(POPL), 1–27 (2019)
31. Ryzhyk, L., Budiu, M.: Differential datalog. Datalog **2**, 4–5 (2019)
32. Schäfer, M., Avgustinov, P., de Moor, O.: Algebraic data types for object-oriented datalog (2017)
33. Schrijver, A.: Theory of Linear and Integer Programming. John Wiley & Sons, Inc., USA (1986)
34. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: A java bytecode optimization framework. In: CASCON First Decade High Impact Papers, pp. 214–224 (2010)

35. Weiser, M.: Program slicing. IEEE Transactions on software engineering (4), 352–357 (1984)
36. Yan, M., Xia, X., Lo, D., Hassan, A.E., Li, S.: Characterizing and identifying reverted commits. Empirical Softw. Engg. **24**(4), 2171–2208 (aug 2019). https://doi.org/10.1007/s10664-019-09688-8, https://doi.org/10.1007/s10664-019-09688-8
37. Yoon, Y., Myers, B.A.: An exploratory study of backtracking strategies used by developers. In: Proceedings of the 5th International Workshop on Co-Operative and Human Aspects of Software Engineering. p. 138–144. CHASE '12, IEEE Press (2012)
38. Zeller, A.: Yesterday, my program worked. today, it does not. why? ACM SIGSOFT Software engineering notes **24**(6), 253–267 (1999)
39. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering **28**(2), 183–200 (2002)
40. Zhao, D., Subotic, P., Raghothaman, M., Scholz, B.: Towards elastic incrementalization for datalog. In: 23rd International Symposium on Principles and Practice of Declarative Programming. pp. 1–16 (2021)
41. Zhao, D., Subotić, P., Scholz, B.: Debugging large-scale datalog: A scalable provenance evaluation strategy. ACM Transactions on Programming Languages and Systems (TOPLAS) **42**(2), 1–35 (2020)
42. Zhou, W., Sherr, M., Tao, T., Li, X., Loo, B.T., Mao, Y.: Efficient querying and maintenance of network provenance at internet-scale. Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data pp. 615–626 (2010)