TPLP: Page 1–25.

doi:10.1017/S147106842500002X

Provenance Guided Rollback Suggestions

DAVID ZHAO
University of Sydney, Glebe, New South Wales, Australia
(*e-mail:* d-z@outlook.com)

PAVLE SUBOTIĆ
Microsoft, Redmond, WA, USA
(*e-mail:* pavlesubotic@microsoft.com)

MUKUND RAGHOTHAMAN
University Southern California, Los Angeles, CA, USA
(*e-mail:* raghotha@usc.edu)

BERNHARD SCHOLZ
University of Sydney, Glebe, New South Wales, Australia
(*e-mail:* bernhard.scholz@sydney.edu.au)

submitted 31 May 2023; revised 9 November 2024; accepted 27 January 2025

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.

KEYWORDS: logic programming methodology and applications

1 Introduction

Datalog has achieved widespread adoption in recent years, particularly in static analysis use cases Grech *et al.* (2018, 2019); Allen *et al.* (2015); Grech *et al.* (2018, 2019); Zhou *et al.* (2010); Huang *et al.* (2011); Backes *et al.* (2019) 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

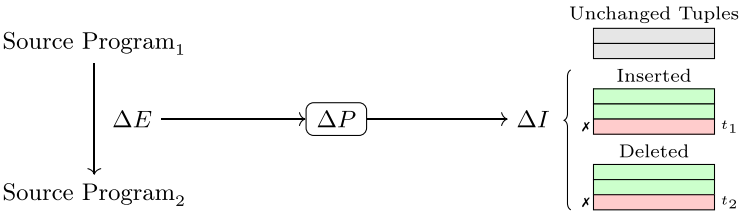

Fig. 1. A scenario where an incremental update results in faults in the output.

made to a code base Distefano *et al.* (2019); Github CodeQL (2021). 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 Zhao *et al.* (2021); Motik *et al.* (2019); Ryzhyk and Budiu (2019); McSherry *et al.* (2013) 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 Yan *et al.* (2019a); Yoon and Myers (2012). 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.

For instance, consider the example in Figure 1. The diagram shows the use of incremental evaluation for program analysis use cases. On the left, the source program is updated, resulting in a change ΔE, which is input to the incremental program analysis ΔP. After computing the incremental update, some result tuples are unchanged, some are inserted, and some are deleted. However, some of the changes (insertions or deletions) may be *unwanted* (i.e., the user does not agree with the change), and hence we can view these as *faults* that appeared as a result of the incremental update.

Existing state-of-the-art Datalog debugging techniques that are available employ data provenance Karvounarakis *et al.* (2010); Zhao *et al.* (2020) or algorithmic debugging Caballero *et al.* (2017) 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* Zeller (1999); Zeller and Hildebrandt (2002), 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 Raghothaman *et al.* (2019) 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

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

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 Jordan *et al.* (2016); Zhao *et al.* (2021) Datalog engine and evaluated its effectiveness on DaCapo Blackburn *et al.* (2006) Java program benchmarks analyzed by the Doop Bravenboer and Smaragdakis (2009) 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

2.1 Motivating example

Consider a Datalog points-to analysis in Figure 2. Here, we show an input program to analyze (Figure 2a), which is encoded as a set of tuples (Figure 2b) by an *extractor*, which maintains a mapping between tuples and source code Jordan *et al.* (2016); Schäfer *et al.* (2017); Vallée-Rai *et al.* (2010). We have relations `new`, `store`, `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, Figure 2c has four rules. A rule is of the form:

$$R_h(X_h) \texttt{ :- } R_1(X_1), \ldots, R_k(X_k).$$

Each $R(X)$ is an *atom*, 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

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

(a)

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

Input Program

(b)

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

store(admin,session,sec).

assign(userSession,ins).
```

EDB Tuples

(c)

```
// 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.
```

Datalog Points-to Analysis

Fig. 2. Program analysis datalog setup.

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 Figure 2a 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

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

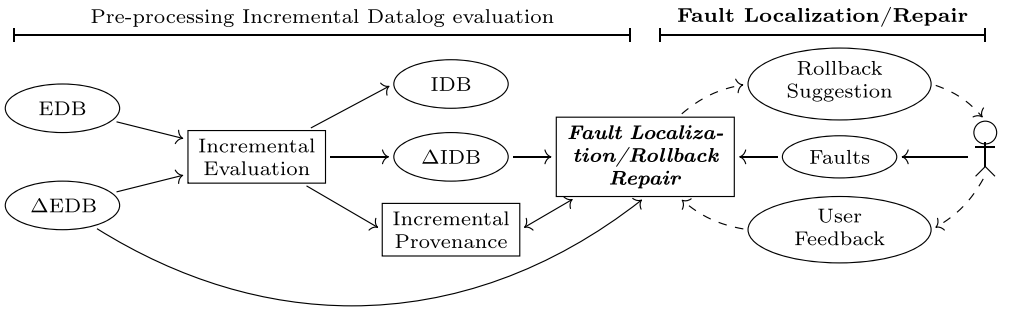

Fig. 3. Fault localization and repair system.

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.

2.1.1 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 3. The first portion of the system is the incremental Datalog evaluation. Here, the incremental evaluation takes an 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.

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

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 $S \subseteq \Delta\text{EDB}$ 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 $S \subseteq \Delta\text{EDB}$ is such that if we were to remove S from ΔEDB, then the resulting diff would *not* produce the faults.

3 Incremental Provenance

3.1 Background

3.1.1 Provenance

Provenance Caballero *et al.* (2017); Zhao *et al.* (2020); Raghothaman *et al.* (2019) 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:

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

This proof tree explains the derivation of `vpt(userSession, L3)` through input and intermediate tuples and which Datalog rules were involved in the derivation. Therefore, these proof trees provide a link between faulty tuples and the structure of the program itself. This forms the basis for our incremental Datalog debugging approach.

A two-phase approach for computing provenance was introduced in Zhao *et al.* (2020). The key idea is to instrument the Datalog program to compute some provenance information alongside the actual tuples, and then to use this information to produce proof trees.

The two-phase process consists of:

1. Instrumented Datalog evaluation: annotating each tuple with *provenance annotations* while computing the IDB. In particular, for each tuple t, the system stores the height of the minimal height proof tree for t. For an EDB tuple, the height is 0, meanwhile the height of an IDB tuple is computed by $h(t) = \max\{h(t_1), \ldots, h(t_k)\} + 1$ if t is derived by a rule instantiation t `:-` $t_1, \ldots, t_k$.
2. Provenance query answering: using the annotated IDB to answer provenance queries of the form of "`explain vpt(userSession, L3)`". Internally, the system generates and solves constraints using the provenance annotations to construct the proof trees one level at a time.

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

For the running example, the tuple `vpt(userSession, L3)` gets a height annotation of 2 during the instrumented evaluation phase. Then, during the query answering, the system generates a constraint saying "find the tuples matching the body of a rule for `vpt` with height less than 2".

3.1.2 Incremental evaluation

Incremental Datalog evaluation provides an efficient method of *updating* the results of a computation given some small changes to the input (where some tuples are inserted and/or deleted). Previous incremental evaluation approaches, such as Delete-Rederive (DRed) Gupta *et al.* (1993), had shortcomings concerning over-deletion and the re-derivation step required to resolve this. However, modern incremental evaluation approaches typically use a counting-based approach. For these strategies, each IDB tuple is associated with a count representing the number of different derivations for that tuple. When an EDB tuple t is inserted or deleted, tuples depending on t have their counts incremented or decremented respectively. If a tuple's count reaches 0, then that tuple is removed from the IDB.

To handle recursion, techniques such as Differential Dataflow McSherry *et al.* (2013) propose storing counters for each tuple *per iteration* of the recursion. Note that if rules are alternately increasing and decreasing, then they would be operating in different iterations. Therefore, each tuple is associated with an iteration number and a count. This increases the space overhead for keeping the auxiliary information but allows for precise recording of the incremental updates in a recursive setting. Variations of this idea, such as Elastic Incremental Evaluation Zhao *et al.* (2021), propose a trade-off of lower space overhead at the cost of requiring some recomputation to maintain precision of the derivation counts.

In our running example, the insertion of two lines in the source program result in the insertion of two EDB tuples for the pointer analysis: `assign(upgradedSession, userSession)` and `load(userSession, admin, session)`.

Using a counting approach Zhao *et al.* (2021); McSherry *et al.* (2013), we can apply incremental counting versions of the Datalog rules to compute an updated IDB. In this instance, we can apply rule r_2 as

```
vpt(upgradedSession,L3)^+1 :-
    assign(upgradedSession,userSession)^+1,
    vpt(userSession,L3).
```

The superscripts denote the changes in count as a result of the newly inserted EDB tuple `assign(upgradedSession,userSession)`. These updated counts are then propagated in subsequent rules, until the IDB has reached fixpoint. Further details are described in Zhao *et al.* (2021).

3.2 Combining provenance and incremental evaluation

For fault localization and rollback, a novel provenance strategy is required that builds on incremental evaluation. *Incremental provenance* restricts the computations of the proof

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

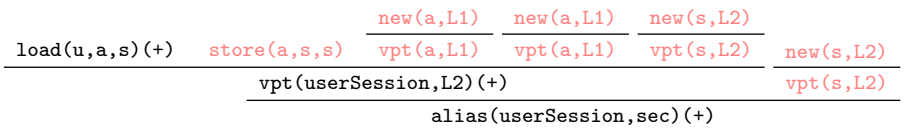

Fig. 4. The proof tree for `alias(userSession,sec)`. `(+)` denotes tuples that are inserted as a result of the incremental update, red denotes tuples that were not affected by the incremental update.

tree to the portions affected by the incremental update only. For example, Figure 4 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.

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

Definition 3.1.
Let P be a Datalog program, E be an EDB, ΔE be an incremental update, and t be a tuple contained in both E and ΔE. Then, $inc - prov(P, E, t, \Delta E)$ is the set of tuples that appear in the proof tree for a tuple t, that are also inserted as a result of ΔE. In the remainder of the paper, we omit P and E if they are unambiguous.

To compute provenance information efficiently in an incremental evaluation setting, we introduce a novel method combining the provenance annotations of Zhao *et al.* (2020) with the incremental annotations of Zhao *et al.* (2021). Recall, from Section 3.1, that provenance annotations include the height of the minimal height proof tree for a tuple, computed by taking the maximum height of all tuples in its derivation. Also recall that incremental annotations include the iteration number in which a tuple is derived.

To combine these two, we can 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 Zhao *et al.* (2020) to use an automated procedure that enumerates exactly the portions of the proof tree that have been affected by the incremental update.

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

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 output* of a program.

Definition 4.1
(Intended Output). *The intended output 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 output for a program, a *fault* can be defined as follows:

Definition 4.2
(Fault). *Let P be a Datalog program, with input set E and intended output (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, that is $t \in I_+ \setminus P(E)$ or t is undesirable but produced, that is $t \in P(E) \cap I_-$.*

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

4.1.1 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 4.3
(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.*

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

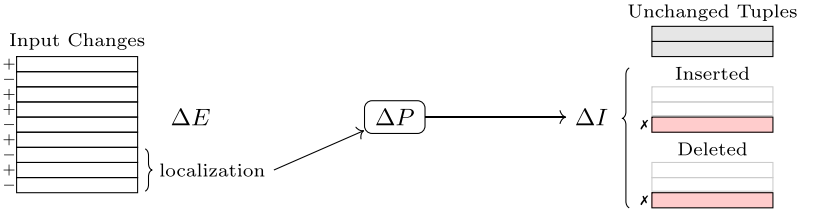

Fig. 5. A fault localization is a subset of input changes such that the faults are still reproduced.

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 $\delta E \subseteq \Delta E_{1\in 2}$ such that $E_1 \uplus \delta E$ produces all $t \in F$

1: **for** tuple $t \in F$ **do**
2: Let `inc-prov`$(P, E2, t, \Delta E_{1\to 2})$ be an incremental proof tree of t containing tuples that were inserted due to $\Delta E_{1\to 2}$
3: **end for**
4: **return** $\cup_{t\in F}(inc - prov(t, \Delta E_{1\to 2}) \cap \Delta E_{1\to 2})$

4.1.2 Rollback suggestion

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

Definition 4.4
(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.

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. On its own, fault localization forms an important part of the reproduction step of any fault investigation. Moreover, it is also fundamental for rollback repair of missing tuples or negations (see Section 4.4).

Consider the example in Figure 5. This diagram illustrates that a fault localization is a subset of the input changes $L \subseteq \Delta E$ such that when L is used as the input changes in the incremental evaluation, the resulting update still produces the faults.

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.

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 $\Delta E_{1\to 2}$

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

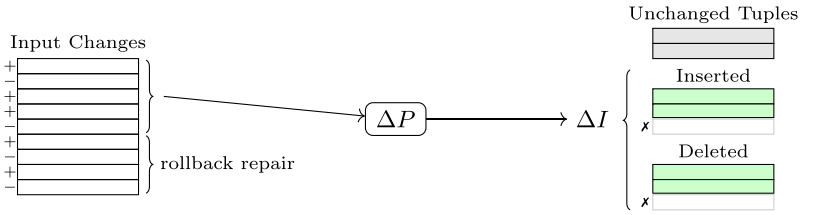

Fig. 6. An input debugging suggestion is a subset of input changes such that the remainder of the input changes no longer produce the faults.

and cause the existence of each fault tuple t. Therefore, by returning the union $\cup_{t \in F}(\texttt{inc-prov}(t, \Delta E_{1\to 2}) \cap \Delta E_{1\to 2})$, the algorithm produces a subset of $\Delta E_{1\to 2}$ that reproduces the faults.

4.3 Rollback repair

A rollback repair is a subset of the input changes such that the remaining changes 'fix' the faults. Consider Figure 6, which shows that a rollback repair is a small subset of the input changes, such that the remainder of the changes no longer produce the faults when used as an incremental update.

The rollback repair algorithm produces a rollback suggestion. As with fault localization, our presentation begins 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.

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) Schrijver (1998) as a vehicle to disable ΔEDB tuples so that the fault tuples vanish in the resulting IDB. We phrase the proof trees as a pseudo-Boolean formula Hooker (1992) whose propositions represent the ΔEDB and IDB tuples in the proof trees. In the ILP, the faulty tuples are constrained to be false, and the ΔEDB tuples assuming the true value are to be maximized, that is we wish to eliminate the least number of tuples in ΔEDB for repair. The ILP created in Alg. 2 introduces a variable for each tuple (either IDB or ΔEDB) that appears in *all* incremental proof trees for the fault tuples. For the ILP model, we have three types of constraints:

1. to encode each one-step proof tree,
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, where $t_1, \ldots, t_k$ and t_h are Boolean variables:

$$\frac{t_1 \quad \ldots \quad t_k}{t_h} \equiv t_1 \wedge \ldots \wedge t_k \implies t_h$$

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

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 $E_1 \uplus (\Delta E_{1\in 2} \backslash \delta E)$ does not produce t_r

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:
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 $t \in F$
for each tuple $t_f \in F$ **do**
    for each proof tree $T_i \in$ `all-inc-prov`$(t_f, \Delta E_{1\to 2})$ **do**
        for each line $t_h \leftarrow t_1 \wedge \ldots \wedge t_k$ in T_i **do**
            Add a constraint $x_{t_1} + \ldots + x_{t_k} - x_{t_h} \leq k - 1$
        end for
    end for
    Add a constraint $x_{t_f} = 0$
end for
Add the objective function maximize $\sum_{t_e \in EDB} x_{t_e}$
Solve the ILP
Return $\{t \in \Delta E_{1\to 2} \mid x_t = 0\}$

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, that is $x_{t_f} = 0$. 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 $\sum_{t \in \Delta E_{1\to 2}} t$.

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

The solution of the ILP permits us to determine the EDB tuples for repair, that is $\{t \in \Delta E_{1\to 2} \mid x_t = 0\}$ 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.

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

4.4 Extensions

4.4.1 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, that is 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 *roll back* 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 $\Delta E_{2\to 1}$. 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 $\Delta E_{1\to 2}$, we compute a solution for the dual problem. The dual problem for localization is to roll back the appearance of t after applying $\Delta E_{2\to 1}$, and similarly, the dual problem for rollback is localization. We note the fault localization on its own is an important part of the investigation process, in case algorithm will compute a fault localization for the diff in the reverse direction.

4.4.2 Stratified negation

Stratified negation is a common extension for Datalog. With stratified negation, atoms in the body of a Datalog rule may appear negated, for example

$$R_h(X_h) \text{ :- } R_1(X_1), \ldots, !R_k(X_k), \ldots, R_n(X_n).$$

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* or *safety*). 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

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

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.

4.4.3 Changes in datalog rules

The algorithms are presented above in the context of localizing or debugging a change to the input tuples. However, with a simple transformation, the same algorithms can also be applied to changes in Datalog rules. For each Datalog rule, introduce a predicate `Rule(i)`, where `i` is a unique number per rule. Then, the unary relation `Rule` can be considered as EDB, and thus the set of rules can be changed by providing a diff containing insertions or deletions into the `Rule` relation. For example, a transformed set of rules may be:

```
P(x, y) :- E(x, y), Rule(1).
P(x, z) :- E(x, y), P(y, z), Rule(2).
```

Then, by including or excluding 1 or 2 in the EDB relation `Rule`, the underlying Datalog rules can be 'switched on or off,' and a change to the Datalog program can be expressed as a diff in the `Rule` relation.

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).

The main part of the algorithm is a worklist loop (lines 5 to 15). 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, that is 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

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

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 output $(I_+,I_,)$, compute a subset $\delta E \subseteq \Delta E_{1\to 2}$ such that $\Delta E_{1\to 2} \setminus \delta E$ satisfies the intended output

1: Let E_2 be the EDB after applying the diff: $E_1 \uplus \Delta E_{1\to 2}$
2: Let F^+ be appearing unwanted faults: $\{I_- \cap P(E_2)\}$
3: Let F^- be missing desirable faults: $\{I_+ \setminus P(E_2)\}$
4: Let L be the set of repair tuples, initialized to $\emptyset$
6: **while** both F^+ and F^- are non-empty **do**
6: Add Rollback-Repair(P, E_2, $\Delta E_{1\to 2}$, F^+) to L
7: **for** negated tuples $!t \in L$ **do**
8: Add t to F^-
9: **end for**
10: Clear F^+
11: Add Localize-Faults(P, E_1, $\Delta E_{2\to 1}$, F^-) to L
12: **for** negated tuples $!t \in L$ **do**P
13: Add t to F^+
14: **end for**
15: Clear F^-
16: **end while**
17: **return** L

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 11.

4.5.1 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 4. 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.

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

4.6 Correctness and Optimality

In this section, we discuss the correctness and optimality of our algorithms. Consider the problem set up with a Datalog program P, an EDB E_1, an incremental update diff $\Delta E_{1\to 2}$, and a set of fault tuples F. For the fault localization algorithm, correctness implies that the result reproduces the faults, that is that $F \subseteq (E_1 \uplus \delta E)$. Meanwhile, the correctness of a rollback repair implies that the result prevents faults from appearing, that is that $(E_1 \uplus (\Delta E_{1\to 2} \setminus \delta_{\times} E)) \cap F = \emptyset$

Optimality is measured by how minimal the solution is. For both fault localization and rollback repair, a solution is optimal if there is no smaller subset of the solution which correctly solves the problem.

4.6.1 Fault localization

The correctness of fault localization (Algorithm 1) lies in the semantics of the proof trees. Consider a single proof tree. If every EDB tuple in the proof tree exists as input, then the resulting tuple at the root of the proof tree would be computed by the Datalog program. Therefore, since the fault localization algorithm returns *all* ΔEDB tuples in the proof tree, then the resulting fault tuple will be in the result.

The optimality of the fault localization result is dependent on the properties of the proof trees produced in the Datalog engine. If these proof trees are minimal in terms of the number of EDB tuples, then the fault localization result will also be minimal in size. This property can be guaranteed depending on how the proof tree generation is implemented (see Zhao *et al.* (2020)), however the details are outside the scope of this paper.

4.6.2 Rollback suggestions

The crucial step in the rollback repair algorithm (Algorithm 2) involves encoding the proof trees as an ILP. These ILP constraints directly encode the logical formulae representing the semantics of the proof trees, with additional constraints asserting that the faulty tuples must be false (i.e., that they are not computed by the EDB satisfying the ILP). Therefore, the correctness of the rollback repair algorithm results from the correctness of the ILP encoding and solving.

To consider the optimality of a rollback suggestion, we first note that the algorithm uses *all* non-cyclic proof trees. This means that the properties of each proof tree do not affect optimality, but rather, the optimality is a result of the maximization constraint in the ILP encoding. This constraint represents that the maximum number of tuples in ΔEDB must be kept in the solution, which is equivalent to saying that the rollback repair includes the *minimum* number of necessary tuples. Hence, the solution is indeed optimal.

4.6.3 Full algorithm

Each component of the full algorithm is correct, as discussed above, and therefore it only remains to be shown that considering the dual problem for negations is correct. This correctness is discussed in Section 4.4, and thus the full algorithm identifies a correct debugging suggestion.

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

However, the full algorithm is not necessarily optimal in the presence of negation. For example, consider when an initial debugging suggestion includes a negated tuple, t. Then, the full algorithm computes the dual problem of localizing the appearance of t with the opposite diff $\Delta E_{2\to 1}$. However, this opposite diff does not consider the initial debugging suggestion (only the negated tuple), and thus, the result may not be optimal. In practice, this sub-optimality rarely affects the solution, and the result is generally optimal or close to optimal.

4.6.4 Complexity

The fault localization algorithm (Algorithm 1) simply computes the provenance for each fault tuple. From Zhao *et al.* (2020), computing a proof tree requires $O(h \log^m |\text{IDB}|)$ time, where h is the height of the proof tree, m is the nesting depth of the joins in the Datalog program, and |IDB| is the total number of tuples computed in the IDB.

The rollback suggestion algorithm computes the full provenance of the fault tuple, which requires up to n applications of the provenance algorithm, if there are n proof trees for the tuple. However, the integer linear programming portion is exponential in complexity, with branch-and-bound-based algorithms Clausen (1999) taking $O(2^{|V|})$ runtime, where $|V|$ is the number of variables. In our case, there is one variable for each tuple in the full provenance, which is up to |IDB| in the worst case. This dominates the runtime of our algorithm, resulting in a total complexity of $O(nh \log^m |\text{IDB}|) + O(2^{|\text{IDB}|})$.

In practice, however, the size of the full provenance for a fault tuple is far smaller than the full IDB, resulting in reasonable real-world performance even for large Datalog programs.

For comparison, the delta debugging approach only needs to check a linear number of subsets of the incremental update. However, for each subset, delta debugging needs to evaluate an (incremental) Datalog program, which is polynomial in complexity, but sometimes prohibitive in practice.

5 Implementation

The implementation of our algorithms first involved extending the Soufflé Datalog engine Jordan *et al.* (2016). Soufflé already includes utilities for computing proof trees (also called provenance) Zhao *et al.* (2020) and incremental evaluation Zhao *et al.* (2021). To support fault localization and rollback suggestions, we extended Soufflé to support *incremental provenance*. This involved interoperability between the provenance and incremental evaluation portions, to allow the provenance mechanism to use the same instrumentation originally designed for incremental evaluation.

We implemented the fault localization and repair algorithms using Python.[1] The implementations of the Fault Localization and Rollback Repair algorithms follow directly from their presentations in this paper. For any operations which require calling a Datalog engine, we call out to our modified Soufflé engine. One potential source of inefficiency in our implementation is that Soufflé does not have direct Python interoperability, so we

[1] Available at github.com/davidwzhao/souffle-fault-localization

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

have to read/write tuples through the filesystem or pipes to interact with Soufflé. For solving integer linear programs, we use the GLPK solver GLPK (GNU Linear Programming Kit) (2012) in the PuLP Python library.

For the full algorithm, we need to compute the dual versions of each problem. For efficiency, we do not construct the full dual version of the problem as they are needed, but instead, we maintain two instances of Soufflé: one for the *forward* problem, and one for the *reverse* direction. Using these two instances of Soufflé, we can easily compute the fault localizations or rollback suggestions as needed, without re-instantiating the Datalog engine.

6 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?

6.1 Experimental setup[2]

Our main point of comparison in our experimental evaluation is the delta debugging approach, such as that used in the ProSynth Datalog synthesis framework Raghothaman *et al.* (2019). 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 Bravenboer and Smaragdakis (2009) with the DaCapo set of Java benchmarks Blackburn *et al.* (2006). 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.

6.1.1 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×[3] compared to 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

[2] We use an Intel Xeon Gold 6130 with 192 GB RAM, GCC 10.3.1, and Python 3.8.10
[3] We say "over" because we bound timeouts to 7200 s.

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

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

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

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 s, 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 s on our benchmarks, with one instance timing out after two hours (7200 s).

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.

6.1.2 Quality

While the delta debugging technique produces 1-minimal results, we observe that despite no overall optimality guarantees, the results show that 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 rollback repair and did not encounter any negations. Therefore, due to the ILP formulation, the results were optimal in these situations. In one case, the rollback repair encountered a negation and flipped to the dual fault localization problem, resulting in the suboptimality. Despite our technique overall not being theoretically optimal, it still produces high-quality results in practice.

7 Related Work

7.1 Logic programming input repair

A plethora of logic programming paradigms exist that can express diagnosis and repair by EDB regeneration Kakas *et al.* (1993); Fan *et al.* (2008a); Gelfond and Lifschitz (1988); El-Hassany *et al.* (2017); Liu *et al.* (2023). 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 Li *et al.* (2018), 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

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

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 Raghothaman *et al.* (2019). 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.

7.2 Database repair

Repairing inconsistent databases with respect to integrity constraints has been extensively investigated in the database community Fan (2009); Bravo and Bertossi (2004); Arenas *et al.* (2003); Fan *et al.* (2008). Unlike our approach, integrity constraints are much less expressive than Datalog; in particular, they do not allow fixpoints in their logic. The technique in Fan *et al.* (2008) shares another similarity in that it also presents repair for incremental SQL evaluation. However, this is limited to relational algebra, that is SQL and Constrained Functional Dependencies (CFDs) that are less expressive than Datalog. A more related variant of database repair is consistent query answering (CQA) Bravo and Bertossi (2004); Arenas *et al.* (2003). 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.

7.3 Program slicing

Program slicing Weiser (1984); Binkley and Gallagher (1996); Ezekiel *et al.* (2021); Harman and Hierons (2001) 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 Cheney (2007), 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.

7.4 Database rollback

Database transaction rollback and partial rollback are well established Mohan *et al.* (1992); Coburn *et al.* (2013) and supported in many DBMS's Oracle rollback (2023); Accelerated database recovery (2023). These techniques often perform rollback for a transaction in the context of data recovery, by logging the effects of each action in the transaction. Techniques such as Antonopoulos *et al.* (2019) improve rollback time by using versioning information. These techniques are limited to SQL transactions while our technique targets recursive datalog queries in an incremental update setting. For the static analysis use case, to the best of our knowledge, we are the first to provide an automated partial commit rollback mechanism based on the analysis output. Nevertheless, it is interesting future work to investigate if our technique can assist in making more efficient partial rollbacks in a DBMS setting.

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

7.5 Automated commit rollback

There is not a lot of work in the literature on automatically detecting and partially rolling back buggy commits, despite several studies Shimagaki *et al.* (2016); Yan *et al.* (2019) highlighting the benefits of identifying such commits and rolling them back as soon as possible. The closest works to ours are techniques Yan *et al.* (2019); Rosen *et al.* (2015); Mockus and Weiss (2000); Kim *et al.* (2008) that seek to identify through statistical models commits that are most likely to be reverted. In contrast, our technique works with a static analyzer that detects bugs in code, and provides users with the option to partially revert the commit so the bug is eliminated.

8 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 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

Accelerated database recovery. 2023. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-accelerated-database-recovery

Allen, N., Scholz, B. and Krishnan, P. 2015. *Staged Points-to Analysis for Large Code Bases.* Springer, Berlin Heidelberg. 131–150.

Antonopoulos, P., Byrne, P., Chen, W., Diaconu, C., Kodandaramaih, R. T., Kodavalla, H., Purnananda, P., Radu, A., Ravella, C. S. and Venkataramanappa, G. M. 2019. Constant time recovery in azure SQL database. *Proceedings of the VLDB Endowment* 12, 12, 2143–2154.

Arenas, M., Bertossi, L. E. and Chomicki, J. 2003. Answer sets for consistent query answering in inconsistent databases. *Theory and Practice of Logic Programming* 3, 4+5, 393–424.

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. and Whaley, B. 2019. Reachability analysis for aws-based networks. In Computer Aided Verification - 31st International Conference, CAV. 2019, New York City, NY, USA, 231–241, July 15-18, 2019, Proceedings, Part II

Binkley, D. W. and Gallagher, K. B. 1996. Program slicing. *Advances in computers* 43, 1–50.

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

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. and Wiedermann, B. 2006. 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, ACM Press, New York, NY, USA, 169–190.

Bravenboer, M. and Smaragdakis, Y. 2009. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not* 44, 10, 243–262.

Bravo, L. and Bertossi, L. E. 2004. Consistent query answering under inclusion dependencies. In Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, October 5–7, 2004. H. Lutfiyya, J. Singer and D. A. Stewart, Eds.Markham, Ontario, Canada, 202–216.

Caballero, R., Riesco, A. and Silva, J. 2017. A survey of algorithmic debugging. *ACM Computing Surveys (CSUR)* 50, 4, 60.

Cheney, J. 2007. Program slicing and data provenance. *IEEE Data Engineering Bulletin* 30, 4, 22–28.

Clausen, J. 1999. *Branch and Bound Algorithms-Principles and Examples*. Department of computer science, University of Copenhagen, 1–30.

Coburn, J., Bunker, T., Schwarz, M., Gupta, R. and Swanson, S. 2013. From aries to mars: Transaction support for next-generation, solid-state drives. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. SOSP '13, Association for Computing Machinery, New York, NY, USA, 197–212.

Distefano, D., Fähndrich, M., Logozzo, F. and O'hearn, P. W. 2019. Scaling static analyses at Facebook. *Communications of the ACM* 62, 8, 62–70.

El-hassany, A., Tsankov, P., Vanbever, L. and Vechev, M. (2017) Network-wide configuration synthesis. In *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 261–281.

Ezekiel, S., Lukas, K., Marcel, B. and Zeller, A. 2021. Locating faults with program slicing: An empirical analysis. *Empirical Software Engineering* 26, pp. 1–45

Fan, W. 2009. *Constraint-Driven Database Repair*, Springer US, Boston, MA, 458–463.

Fan, W., Geerts, F. and Jia, X. 2008. Semandaq: A data quality system based on conditional functional dependencies. *Proceedings of the VLDB Endowment* 1, 2, 1460–1463.

Gelfond, M. and Lifschitz, V. 1988. *The Stable Model Semantics for Logic Programming*. MIT Press, 1070–1080.

Github codeql. 2021. https://codeql.github.com/ Accessed: 19-10-2021.

GLPK (gnu linear programming kit). 2012. https://www.gnu.org/software/glpk/glpk.html

Grech, N., Brent, L., Scholz, B. and Smaragdakis, Y. 2019. Gigahorse: Thorough, declarative decompilation of smart contracts. In Proceedings of the 41th International Conference on Software Engineering, ICSE 2019, ACM, Montreal, QC, Canada. (to appear)

Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B. and Smaragdakis, Y. (2018) Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Communications of the ACM* 63, 10, 87–95.

Gupta, A., Mumick, I. S. and Subrahmanian, V. S. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2, 157–166.

Harman, M. and Hierons, R. 2001. An overview of program slicing. *Software Focus* 2, 3, 85–92.

Hooker, J. 1992. Generalized resolution for 0-1 linear inequalities. *Annals of Mathematics and Artificial Intelligence* 6, 1–3, 271–286.

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press

Huang, S. S., Green, T. J. and Loo, B. T. 2011. Datalog and emerging applications: An interactive tutorial. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. SIGMOD '11, ACM, 1213–1216.

Jordan, H., Scholz, B. and Subotić, P. 2016. Soufflé: On synthesis of program analyzers. In International Conference on Computer Aided Verification, Springer, 422–430.

Kakas, A. C., Kowalski, R. A. and Toni, F. (1993). Abductive logic programming. *Journal of logic and computation*, 2(6), pp. 719–770.

Karvounarakis, G., Ives, Z. G. and Tannen, V. (2010). Querying data provenance. In SIGMOD '10, Association for Computing Machinery, New York, NY, USA, 951–962.

Kim, S., Whitehead, E. J. and Zhang, Y. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2, 181–196.

Li, X., Bundy, A. and Smaill, A. 2018. Abc repair system for datalog-like theories. In 10th International Conference on Knowledge Engineering and Ontology Development, SCITEPRESS, 335–342.

Liu, Y., Mechtaev, S., Subotic, P. and Roychoudhury, A. 2023. Program repair guided by datalog-defined static analysis. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE. 2023, S. Chandra, K. Blincoe and P. Tonella, ACM, San Francisco, CA, USA, 1216–1228, December 3-9, *2023*

Mcsherry, F., Murray, D. G., Isaacs, R. and Isard, M. 2013. Differential dataflow. In 6th Biennial Conference on Innovative Data Systems Research.

Mockus, A. and Weiss, D. M. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2, 169–180.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H. and Schwarz, P. 1992. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems* 17, 1, 94–162.

Motik, B., Nenov, Y., Piro, R. and Horrocks, I. 2019. Maintenance of datalog materialisations revisited. *Artificial Intelligence* 269, 76–136.

Oracle rollback. 2023. https://docs.oracle.com/cd/B10500_01/server.920/a96533/instreco.htm#429546

Raghothaman, M., Mendelson, J., Zhao, D., Naik, M. and Scholz, B. 2019. Provenance-guided synthesis of datalog programs. In Proceedings of the ACM on Programming Languages, Vol. 4, 1–27.

Rosen, C., Grawi, B. and Shihab, E. 2015. Commit guru: Analytics and risk prediction of software commits. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA, 966–969.

Ryzhyk, L. and Budiu, M. 2019. Differential datalog. *Datalog* 2, 4–5.

Schäfer, M., Avgustinov, P. and De moor, O. (2017). Algebraic data types for object-oriented datalog [online].

Schrijver, A. 1998. *Theory of Linear and Integer Programming*, John Wiley & Sons Inc., USA.

ShimagakI, J., Kamei, Y., Mcintosh, S., Pursehouse, D. and Ubayashi, N. 2016. Why are commits being reverted?: A comparative study of industrial and open source projects. In 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), 301–311. IEEE.

Vallée-rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V. 2010. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, 214–224.

Weiser, M. 1984. Program slicing. *IEEE Transactions on Software Engineering* 4, 4, 352–357.

Yan, M., Xia, X., Lo, D., Hassan, A. E. and Li, S. 2019. Characterizing and identifying reverted commits. *Empirical Software Engineering* 24, 4, 2171–2208.

Yoon, Y. and Myers, B. A. 2012. 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. CHASE '12, IEEE Press, 138–144.

Zeller, A. (1999) Yesterday, my program worked. today, it does not. In *Why? ACM SIGSOFT Software Engineering Notes 24*, Vol. 6, 253–267.

Zeller, A. and Hildebrandt, R. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2, 183–200.

Zhao, D., Subotic, P., Raghothaman, M. and Scholz, B. 2021. Towards elastic incrementalization for datalog. In 23rd International Symposium on Principles and Practice of Declarative Programming, 1–16.

Zhao, D., Subotić, P. and Scholz, B. 2020. Debugging large-scale datalog: A scalable provenance evaluation strategy. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42, 2, 1–35.

Zhou, W., Sherr, M., Tao, T., Li, X., Loo, B. T. and Mao, Y. 2010. Efficient querying and maintenance of network provenance at internet-scale. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, 615–626.

https://doi.org/10.1017/S147106842500002X Published online by Cambridge University Press