Metamorphic Testing of Datalog Engines

Muhammad Numair Mansur
MPI-SWS, Germany
numair@mpi-sws.org

Maria Christakis
MPI-SWS, Germany
maria@mpi-sws.org

Valentin Wüstholz
ConsenSys, Germany
valentin.wustholz@consensys.net

ABSTRACT

Datalog is a popular query language with applications in several domains. Like any complex piece of software, Datalog engines may contain bugs. The most critical ones manifest as incorrect results when evaluating queries—we refer to these as *query bugs*. Given the wide applicability of the language, query bugs may have detrimental consequences, for instance, by compromising the soundness of a program analysis that is implemented and formalized in Datalog. In this paper, we present the first metamorphic-testing approach for detecting query bugs in Datalog engines. We ran our tool on three mature engines and found 13 previously unknown query bugs, some of which are deep and revealed critical semantic issues.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

metamorphic testing, fuzzing, Datalog

ACM Reference Format:
Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic Testing of Datalog Engines. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468573

1 INTRODUCTION

Datalog [28] is a declarative, logic-based query language that is syntactically a subset of Prolog. Datalog is expressive, yet concise, and as a result, it is used as a domain-specific language in several application domains, such as natural-language processing [50], bioinformatics [38, 61], big-data analytics [29, 31], networking [46], program analysis [11, 22, 26, 51, 74], robotics [53], generic graph databases [64], and security [12, 13, 27, 69].

Query evaluation is performed by *Datalog engines*, prominent examples of which include Soufflé [32], bddbddb [73], DDlog [59], μZ [30], and LogicBlox [3]. However, as any complex piece of software, Datalog engines may contain bugs, resulting in incorrect query results. An incorrect result may manifest by including wrong entries or by missing entries that should have been included. We refer to such bugs as *query bugs*.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ESEC/FSE '21, August 23–28, 2021, Athens, Greece
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468573

Depending on the application domain, query bugs may have detrimental consequences. In particular, when a buggy Datalog engine is used in program analysis, it could compromise *soundness* of the verification process; in other words, it could cause an analyzer to verify incorrect software. As an example, imagine a static analyzer that uses Datalog to implement a may-alias (or must-alias) analysis. A query bug that results in computing fewer (or more) aliases could lead to missing critical bugs in the analyzed software.

In this paper, we present the *first* automatic test-case generation approach for detecting query bugs in Datalog engines. A major challenge in finding such bugs is the lack of an *oracle* specifying *expected* query results. This problem may be overcome with a technique known as *differential testing* [48]. Differential testing would involve running multiple Datalog engines on a common set of programs and comparing their results for discrepancies. In our context, this would be extremely difficult as there exists no unified standard for Datalog syntax; as a result, many different dialects have emerged.

Our approach circumvents the lack of an oracle using an alternative technique, namely *metamorphic testing* [21]. It works by transforming a Datalog program such that the new result has an a-priori known relationship to the result of the original program. Examples of such a relationship are that the new result should be equivalent to the original, contained in the original, or containing the original. To ensure that these oracles are known in advance, we design metamorphic transformations based on database theory, and in particular, formal properties of *conjunctive queries*.

Despite their simplicity, conjunctive queries constitute an important class of database queries due to their theoretical properties. Specifically, while many fundamental problems in query optimization and minimization are computationally hard—or even undecidable—for general forms of queries, they are feasible for conjunctive queries. An example of such a problem is *query containment*, which we discuss in Sect. 3. The key insight behind our approach is to leverage properties of conjunctive queries to develop metamorphic transformations for full-blown Datalog programs.

We implement our approach in a tool called queryFuzz, which we use to test three mature Datalog engines. Not only did we find previously unknown query bugs in all engines, but we also detected 81% of all reported query bugs since May 2020. Moreover, as we describe in Sect. 7, some of these bugs were hidden deep in the engine stack and revealed critical semantic issues.

Contributions. Our paper makes the following contributions:

(1) We present the first metamorphic-testing approach for detecting query bugs in Datalog engines.
(2) We implement our approach in an open-source tool[1], queryFuzz. We are already working closely with the developers

[1] https://github.com/Practical-Formal-Methods/queryFuzz

```
1  // declarations
2  edge(X:number, Y:number).
3  reachable(X:number, Y:number).
4  .output reachable
5
6  // facts
7  edge(1,2).
8  edge(2,3).
9  edge(4,2).
10 edge(2,5).
11
12 // rules
13 reachable(X,Y) :- edge(X,Y).
14 reachable(X,Z) :- edge(X,Y), reachable(Y,Z).
```

Figure 1: A simple Datalog program.

of the mature Datalog engine Soufflé in order to integrate queryFuzz in their development cycle.

(3) We evaluate the effectiveness of queryFuzz by testing three popular Datalog engines. Our tool detected 13 previously unknown query bugs in all three engines as well as many other bugs as a by-product.

Outline. The next section gives an overview of our approach. Sect. 3 provides background on properties of conjunctive queries, Sect. 4 explains the technical details of our approach for these queries, and Sect. 5 generalizes the approach to full-blown Datalog programs. In Sect. 6, we describe the implementation of queryFuzz. We present our experimental evaluation in Sect. 7, discuss related work in Sect. 8, and conclude in Sect. 9.

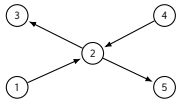
2 OVERVIEW

Datalog is a logic programming language where programs comprise a finite set of *rules* over *relations*. *Input relations* are given in the form of *facts*; they are also commonly referred to as *extensional database (EDB) relations*. *Intensional database (IDB) relations* are defined by logic rules, and one of them is specified as *output*. Fig. 1 shows an example of a simple Datalog program. The rules on lines 13 and 14 define IDB relation `reachable`, which is specified as output on line 4 and computes the transitive closure of input relation `edge`.

Pictorially, `edge` represents the graph in Fig. 2a. There is an edge from node x to node y if `edge(`x`,`y`)` is a fact. Execution of this program is essentially a sequence of derivations, where each step adds an edge tuple to the output relation until a fixed point is reached. Fig. 2b shows the final tuples in `reachable`.

Approach. Using the above example as seed, we now give an overview of our metamorphic-testing approach for Datalog engines. Fig. 3 illustrates its main stages.

The first stage, *Program Generation*, generates a diverse set of programs to be transformed. It takes as input a (possibly empty) seed program, such as that of Fig. 1, and outputs a new program. In case the seed is empty, the new program is randomly generated based on a Datalog grammar. If the seed is not empty, this stage automatically extends it with randomly generated IDB relations using both existing and newly generated facts and rules (again based on a grammar). This is essentially a generalization of the above case where the seed is empty. One of the program relations is then specified as output.



(a) Pictorial view

```
(1,2),(1,3),(1,5),(2,3)
(2,5),(4,2),(4,5),(4,3)
```

(b) Transitive closure

Figure 2: Pictorial view and transitive closure of `edge`.

The second stage, *Program Transformation*, applies metamorphic transformations to the newly generated program (or directly to the seed if the first stage is skipped). These transformations change rules of the program such that—when computing its output using a Datalog engine—the new result has an a-priori known relationship to the old result. In particular, the new result may contain the old one (as computed by program `exp.dl` in Fig. 3), it may be equivalent to the old one (as computed by `equ.dl`), or it may be contained in the old result (as computed by `con.dl`). For example, a transformation in which the new result should be equivalent to the old one is changing line 13 of Fig. 1 to the following:

```
reachable(X,Y) :- edge(X,Y), edge(W,Y).
```

As we will see in the next section, this change appears to be introducing a join, which however has no effect on the result. Another transformation could be applied to line 14 as follows:

```
reachable(X,Z) :- edge(X,X), reachable(X,Z).
```

In this case, the new result should be contained in the old one—in fact, the new result should be empty as there are no edges from a node to itself.

Finally, the third stage, *Bug Detection*, uses these relationships between new and old results (shown in blue and yellow, respectively, in Fig. 3) as *oracles* in order to detect query bugs in the underlying Datalog engine. For instance, imagine that, after transforming line 13 of Fig. 1 as described above, the Datalog engine returns all but one of the tuples shown in Fig. 2b. Since this transformation ensures that the new result is equivalent to the old one, a query bug has been detected. Note that a query bug is also detected if the old result is incorrect as long as the expected relationship to the new result does not hold.

Query bugs. In the rest of this section, we present two query bugs detected by queryFuzz in existing Datalog engines. We provide a complete list of detected bugs and more details in Sect. 7.

Fig. 4 shows a program snippet that was generated by queryFuzz in order to test μZ [30], the Datalog engine of the Z3 SMT solver [23] supporting the bddbddb [73] dialect. Relation `r` (line 4) is defined to compute all tuples in `in2` whose second element is in `in1`. Tuple `(25,10)` is the only one that satisfies this definition. Output relation `out` (line 5) obtains the first element of each tuple in `r`, that is, it computes `25`. This is also the result that is returned by μZ. Now, consider the following transformation applied by queryFuzz to line 5:

```
out(F) :- r(F,C), r(F,A), r(F,B).
```

The result of the new program should still be `25`, but μZ returns values `7–63`. We reported this bug on Z3's GitHub issue tracker[2], and it was immediately confirmed and fixed. In fact, a Z3 developer

[2] https://github.com/Z3Prover/z3/issues/4870

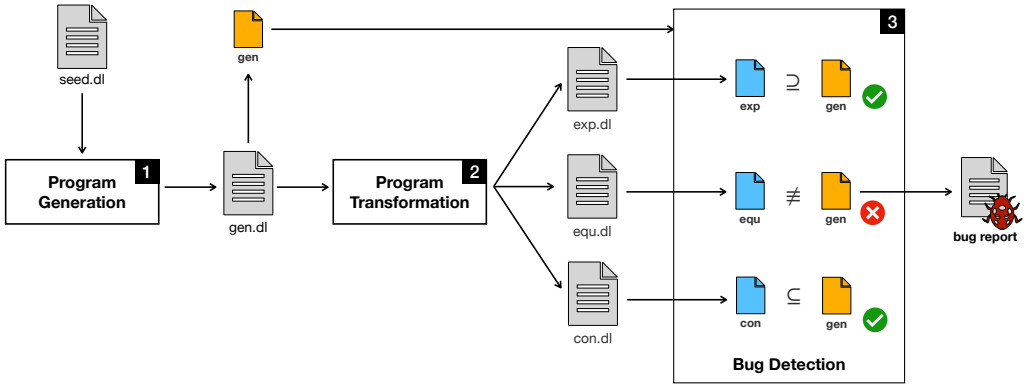


Figure 3: Overview of our approach.

```
1 in1(49). in1(10).
2 in2(25,10). in2(16,13). in2(24,22).
3
4 r(V,M) :- in2(V,M), in1(M).
5 out(F) :- r(F,C).
```

Figure 4: Generated program snippet for testing μZ.

```
1 HqV(a) :- MZV(a,b), MZV(c,d).
2 gQk(jW) :- MZV(jW,jW).
3 QOq(aS,GF) :- MZV(GF,GF), gQk(M), HqV(aS), MZV(aS,M).
4 RwL(qr) :- QOq(u,qr), gQk(u), gQk(u).
5 out(jB,ym) :- gQk(h), RwL(ym), MZV(h,jB).
```

Figure 5: Generated program snippet for testing Soufflé.

commented: "*These are good latent bugs. They exercise some edge cases that slipped through the cracks until now.*"

The code snippet in Fig. 5 was also generated by queryFuzz, this time when testing the Soufflé Datalog engine [32]. Relation `out` is the output relation of the program. When line 1 is changed to

```
Hqv(a) :- MZV(a,b).
```

the program result should remain the same. However, we found that the result of the original program contained 240 entries, whereas that of the transformed program contained 306. We reported this query bug[3], which was immediately fixed.

These types of bugs, detected by queryFuzz, are extremely difficult for unsuspecting users to notice and might compromise upstream applications that rely on a Datalog engine.

3 BACKGROUND

In this section, we review key concepts from database theory, and in particular query optimization, that form the basis of our metamorphic transformations.

A *database schema* $\mathcal{R}$ is a set of relations R. The *arity* of a relation is the number of attributes in the relation. For example, `edge` and `reachable` in Fig. 1 are relations of arity 2. An attribute in a relation can take values from a domain D. Let R be a relation of arity m. A *fact* over R is an expression of the form $R(a_1, ..., a_m)$, where $a_i \in D_i$ for every $i = 1, \ldots, m$, e.g., `edge(1,2)` in Fig. 1. An *instance* of relation R is a finite set of facts over R. A *database instance* I over a database schema $\mathcal{R}$ is a collection of relational instances over the relations $R \in \mathcal{R}$.

A *conjunctive query* (CQ) is a single non-recursive function-free Horn rule, e.g., every rule in Figs. 4 and 5 is a CQ. This is the simplest type of query that can be expressed over a database schema. Syntactically, a conjunctive query Q is an expression of the form

$$P(\vec{U}) \leftarrow R_1(\vec{U_1}), \ldots, R_n(\vec{U_n})$$

where $\vec{U}$ and $\vec{U_i}$ $(1 \leq i \leq n)$ are vectors of variables and constants. Any variable appearing in $\vec{U}$ must also appear in some $\vec{U_i}$. The expression to the left of $\leftarrow$ is the *head* of the query, and the expression to the right is the *body*. Each $R_i(\vec{U_i})$ in the body of the query is a *subgoal*, and $R_i \in \mathcal{R}$ is a *relation*. Note that subgoals can refer to the same relation. The set of answers for query Q w.r.t a database instance I is denoted by $Q(I)$. Given two syntactically different CQs, we now define query *equivalence* and *containment*.

Definition 1 (Query Equivalence). *Two conjunctive queries Q_1 and Q_2 are equivalent, denoted by $Q_1 \equiv Q_2$, iff for every database instance I, we have $Q_1(I) = Q_2(I)$.*

Definition 2 (Query Containment). *Conjunctive query Q_1 is contained in conjunctive query Q_2, denoted by $Q_1 \subseteq Q_2$, iff for every database instance I, we have $Q_1(I) \subseteq Q_2(I)$.*

It is straightforward to see that if $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$, then $Q_1 \equiv Q_2$. A decidable procedure for checking query containment [19] involves determining whether there exists a so-called *containment mapping* between two queries.

Definition 3 (Substitution). *A substitution θ is a mapping from a set of variables V to a set of variables V'.*

Definition 4 (Containment Mapping). *A substitution θ is a containment mapping from conjunctive query Q_2 to conjunctive query Q_1, if Q_2 can be transformed by means of θ to become Q_1.*

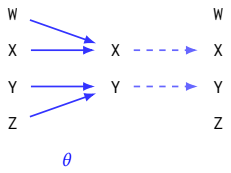
Formally, given two CQs

$$P(\vec{U}) \leftarrow R_1(\vec{U_1}), \ldots, R_n(\vec{U_n}) \qquad (Q_1)$$
$$P'(\vec{V}) \leftarrow S_1(\vec{V_1}), \ldots, S_m(\vec{V_m}) \qquad (Q_2)$$

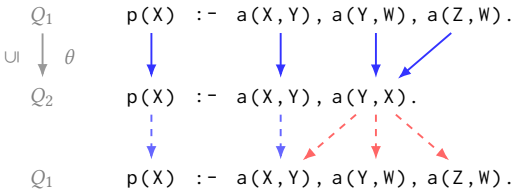
θ is a containment mapping from Q_2 to Q_1 if:

(1) $\theta(P'(\vec{V})) = P(\vec{U})$, and
(2) $\forall i \in \{1, \ldots, m\} \cdot \exists j \in \{1, \ldots, n\} \cdot \theta(S_i(\vec{V_i})) = R_j(\vec{U_j})$.

[3] https://github.com/souffle-lang/souffle/issues/1453



(a) Containment mapping θ from Q_1 to Q_2.



(b) Mapping of head and subgoals induced by θ.

Figure 6: Containment mapping θ from Q_1 to Q_2 induces a mapping of subgoals. No mapping exists from Q_2 to Q_1.

In words, a containment mapping maps variables of Q_2 to variables of Q_1 such that

(1) the head of Q_2 becomes the head of Q_1, and
(2) each subgoal of Q_2 becomes *some* subgoal of Q_1.

Theorem 1. *Let Q_1 and Q_2 be conjunctive queries. Q_2 is contained in Q_1 ($Q_2 \subseteq Q_1$) iff there exists a containment mapping from Q_1 to Q_2.*

As an example, consider the two CQs below (in Datalog syntax):

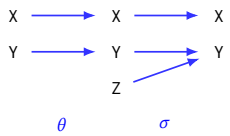
```
p(X) :- a(X,Y), a(Y,W), a(Z,W). // Q1
p(X) :- a(X,Y), a(Y,X).         // Q2
```

Q_2 is contained in Q_1 ($Q_2 \subseteq Q_1$) because there exists a containment mapping θ from Q_1 to Q_2 (shown using solid arrows in Fig. 6a; dotted arrows should be ignored for now). This is indeed a containment mapping because the head of Q_1 is the head of Q_2 and each subgoal of Q_1 becomes a subgoal of Q_2 (shown using solid arrows in Fig. 6b). On the other hand, Q_1 is not contained in Q_2 ($Q_1 \not\subseteq Q_2$) because there does not exist a containment mapping from Q_2 to Q_1, shown with dotted arrows in the figure. If X and Y are mapped to themselves (see Fig. 6a), then the head and first subgoal of Q_2 become the head and first subgoal of Q_1, but the second subgoal of Q_2 cannot become any subgoal of Q_1 (see Fig. 6b; red dotted arrows denote invalid subgoal mappings).

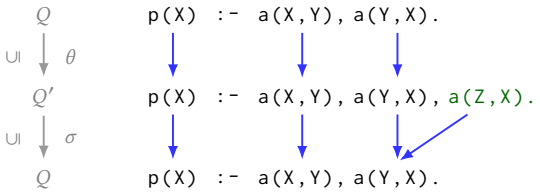
4 METAMORPHIC TRANSFORMATIONS

Using the equivalence and containment properties of CQs, we now present their metamorphic transformations. Note that, in this section, we keep the presentation simple by describing a single transformation to a single conjunctive query. In practice however, our approach can perform sequences of transformations to multiple, more general queries (see Sects. 4.4 and 5 for more details).

Since any conjunctive query may be expressed as a Datalog rule, we refer to CQs as rules in the following. Our metamorphic rule transformations are categorized into three *types*:



(a) Containment mapping θ from Q to Q' and mapping σ from Q' to Q.



(b) Mapping of head and subgoals induced by θ and σ.

Figure 7: Example of AddEqu transformation.

Addition (Add): Rule Q is transformed into $\text{Add}(Q) = Q'$ by adding a subgoal.
Modification (Mod): Rule Q is transformed into $\text{Mod}(Q) = Q'$ by modifying a variable.
Removal (Rem): Rule Q is transformed into $\text{Rem}(Q) = Q'$ by removing a subgoal.

Each of these transformation types may result in any of the following three *outcomes*:

Expansion (Exp): Original rule Q is contained in transformed rule Q', i.e., $Q \subseteq Q'$.
Equivalence (Equ): Original rule Q is equivalent to transformed rule Q', i.e., $Q \equiv Q'$.
Contraction (Con): Transformed rule Q' is contained in original rule Q, i.e., $Q' \subseteq Q$.

We refer to these outcomes as *oracles*.

Based on the above, a *rule transformation* combines a transformation type with an oracle. For instance, AddCon refers to adding a subgoal to a rule Q such that the resulting rule Q' is contained in Q. Next, we describe these transformations in detail.

4.1 Add Transformations

The Add transformations add a subgoal $R(v_1, \ldots, v_n)$ to a rule Q, where $v_1, \ldots, v_n$ are variables—we ignore constants for simplicity.

AddExp. The AddExp transformation ensures that Q is contained in the resulting rule Q', i.e., $Q \subseteq Q'$. However, note that it is not possible to obtain a Q' such that $Q \subset Q'$ by adding a subgoal. The reason is that, when adding a subgoal to Q, there is *always* a containment mapping from Q to Q', i.e., $Q' \subseteq Q$. This is because the head of Q is the head of Q', and each subgoal of Q is in Q'. Consequently, even if there existed a containment mapping in the desirable direction, i.e., $Q \subseteq Q'$, then the two queries would be equivalent, a case that is already covered by AddEqu.

AddEqu. Given that a containment mapping from Q to Q' always exists, the AddEqu transformation guarantees that $Q \equiv Q'$ by ensuring there also exists a containment mapping from Q' to Q. Intuitively, AddEqu adds a new subgoal to Q while avoiding introducing new joins among the existing subgoals, thus preserving the original result. To ensure the existence of a containment mapping

Algorithm 1: ADD transformations

```
1  procedure AddEqu(Q)
2    head, body ← Q
3    g ← RandSubgoal(body)
4    n ← Arity(g)
5    m ← RandIntRange(1, n)
6    for (i ← 0, i < m, i++) do
7      j ← RandIntRange(0, n − 1)
8      g.args[j] ← FreshVar(Q)
9    return head ← body, g.
10 procedure AddCon(Q, relations)
11   g.rel ← RandRelation(relations)
12   n ← Arity(g)
13   vars ← ExtractAllVars(Q)
14   for (i ← 0, i < n, i++) do
15     g.args[i] ← RandVar(vars)
16   head, body ← Q
17   if g ∈ body then
18     return none
19   return head ← body, g.
```

from Q' to Q when adding a subgoal $R(v_1, \ldots, v_n)$ to Q, relation R must already exist in the body of Q.

Example. Fig. 7 shows an example of an AddEqu transformation. The new subgoal `a(Z,X)` (shown in green) maps to `a(Y,X)` when respecting the containment mapping σ from Q' to Q. Although it might appear that the new subgoal introduces a join, this join does not restrict the original result (as computed by the original subgoals) any further.

Algorithm. The algorithm performing this transformation is shown in procedure AddEqu of Alg. 1. First, we extract the head and body of rule Q (line 2). Then, a random subgoal g and its arity n are retrieved from *body* (lines 3–4). On lines 5–8, we replace each of m variables in g with a fresh variable, where m is a random number from 1 to n. Each call to function FreshVar returns a new variable that is not already present in Q. This guarantees that no new joins are introduced. Subgoal g is finally appended to *body*, and new rule Q' is returned (line 9). In the example of Fig. 7, we replace variable `Y` in subgoal `a(Y,X)` of Q with fresh variable `Z` and append this new subgoal to Q in order to generate Q'.

AddCon. The AddCon transformation ensures that rule Q' is contained in original rule Q, i.e., $Q' \subseteq Q$. Intuitively, AddCon adds a new subgoal to Q introducing new joins, thus potentially contracting the original result. To differentiate this transformation from AddEqu, we ensure that a containment mapping does not exist from Q' to Q, i.e., $Q \nsubseteq Q'$. Note, however, that the absence of such a mapping does not mean that Q' produces a *strictly* contracted result. In other words, $Q' \subset Q$ does not always hold; for example, for an empty database instance, the result of Q' is still equivalent to that of Q. To ensure the absence of a containment mapping from Q' to Q when adding a subgoal $R(v_1, \ldots, v_n)$ to Q, relation R must either not already exist in the body of Q, or if it does, its variables should prevent it from being mapped to any subgoal in Q.

Example. Fig. 8 shows an example of an AddCon transformation. The new subgoal `a(Y,Y)` (shown in green) corresponds to relation `a`, which already appears in the body of Q. Despite this, the new

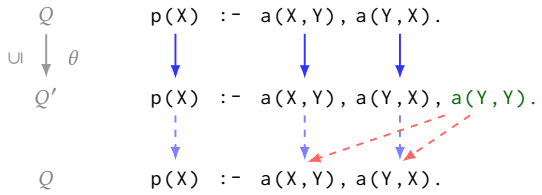
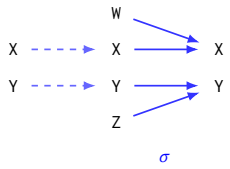
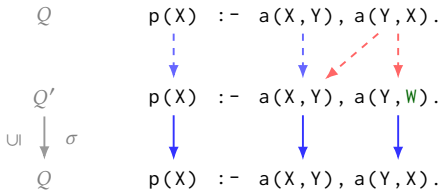


Figure 8: Example of AddCon transformation.



(a) Containment mapping σ from Q' to Q.



(b) Mapping of head and subgoals induced by σ.

Figure 9: Example of ModExp transformation.

subgoal does not map to any subgoal in Q since variable `Y` may not be mapped to both `X` and `Y`.

Algorithm. The algorithm is shown in procedure AddCon of Alg. 1. As a first step, we create a subgoal g by randomly selecting a relation from the set of all relations in the program (line 11). On line 12, we retrieve its arity n. Then, all variables of query Q are extracted in *vars* (line 13), and we initialize each argument of g with a random variable from *vars* (lines 14–15). Using variables in Q for this initialization guarantees that new joins are introduced unless g already appears in *body*. If so, we discard it (lines 17–18), otherwise, we append g to *body* and return new rule Q' (line 19). Note that, when `none` is returned, our implementation tries again. In the example of Fig. 8, we select relation `a`, initialize its arguments with variable `Y`, and append this new subgoal to Q.

4.2 Mod Transformations

The Mod transformations modify a rule Q by renaming a variable appearing in its subgoals.

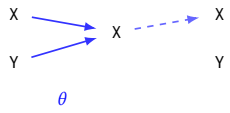
ModExp. Intuitively, this transformation expands the result of Q by renaming a variable in a way that removes existing joins. This is achieved by creating a surjective containment mapping from Q' to Q, i.e., $Q \subseteq Q'$. Note that the mapping may not be bijective as this would make ModExp equivalent to ModEqu.

Example. Fig. 9 shows an example of a ModExp transformation, where variable `X` of subgoal `a(Y,X)` is renamed to `W`.

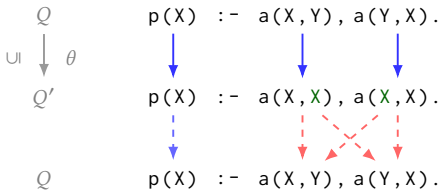
Algorithm. The algorithm for this transformation is shown in procedure ModExp of Alg. 2. We first extract variables *vars* that appear more than once in *body* of rule Q (line 3). A random variable v from *vars* is selected (line 4), and we replace a random occurrence

Algorithm 2: ModExp transformations

```
1  procedure ModExp(Q)
2    head, body ← Q
3    vars ← ExtractReusedVars(body)
4    v ← RandVar(vars)
5    body′ ← ReplaceRandOccurrence(body, v, FreshVar(Q))
6    return head ← body′
7  procedure ModCon(Q)
8    vars ← ExtractAllVars(Q)
9    if |vars| < 2 then
10     return none
11   v ← RandVar(vars)
12   w ← RandVar(vars \ {v})
13   Q′ ← ReplaceVar(Q, v, w)
14   return Q′
```



(a) Containment mapping θ from Q to Q'.



(b) Mapping of head and subgoals induced by θ.

Figure 10: Example of ModCon transformation.

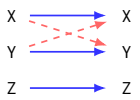
of v in *body* with a fresh variable to get *body*′ (line 5). Replacing an occurrence of a reused variable with a fresh one guarantees that existing joins are removed. Finally, we return *head* ← *body*′ as transformed rule Q' (line 6). In the example of Fig. 9, we choose variable `X`, which appears twice in the body of Q, and replace its second occurrence with fresh variable `W`.

ModEqu. The ModEqu transformation ensures that the result of Q is equivalent to that of Q' by creating a bijective containment mapping between the two rules. A way to guarantee the existence of such a mapping is by replacing *all* occurrences of a variable in Q with those of a fresh variable. Note that this is a very simple transformation, which we include here mainly for completeness.

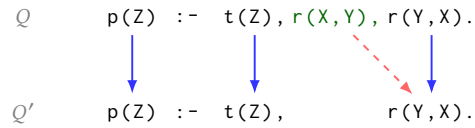
ModCon. Analogously to the ModExp transformation, ModCon renames a variable in Q such that there exists a surjective (and not bijective) containment mapping from Q to Q'.

Example. Fig. 10 shows an example of a ModCon transformation, where all occurrences of variable `Y` are renamed to `X`.

Algorithm. The algorithm is shown in procedure ModCon of Alg. 2. As a first step, we extract all variables *vars* in Q (line 8). If there are fewer than two, we return `none` (lines 9–10). Otherwise, two (different) variables v and w are randomly selected from *vars* (lines 11–12), and we replace all occurrences of v in Q with w to get Q' (line 13). This ensures that new joins are introduced.



(a) No containment mapping from Q to Q'.



(b) Dropped subgoal may not be mapped to any subgoal in Q'.

Figure 11: Example of RemExp transformation.

4.3 Rem Transformations

The Rem transformations remove a subgoal $R(v_1, \ldots, v_n)$ from a rule Q. Analogously to the Add transformations, when removing the subgoal, there is *always* a containment mapping σ from Q' to Q, i.e., $Q \subseteq Q'$. This is because the head of Q' is the head of Q, and each subgoal of Q' is in Q.

RemExp. This transformation checks the existence of a containment mapping from Q to Q'. If such a mapping does *not* exist, then $Q' \not\subseteq Q$ and $Q \subseteq Q'$ (due to σ), that is, the result of Q is expanded. Note that, in general, the problem of checking query containment is NP-complete. However, we can design a containment checker with linear-time complexity because Q' is derived from Q by removing one subgoal. Therefore, it is only necessary to check whether this subgoal of Q may be mapped to any subgoal of Q'.

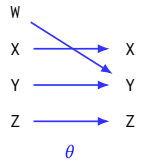
Example. Consider the example in Fig. 11. Removing the second subgoal of Q (shown in green) prevents the existence of a mapping from Q to Q' since it would require each of the variables `X` and `Y` of Q to be mapped to more than one variable of Q'. Consequently, this is a successful RemExp transformation.

Algorithm. The algorithm for this transformation is shown in procedure RemExp of Alg. 3. First, we randomly select a subgoal g from the body of Q (line 13) and remove it to get Q' (line 15). We then check the existence of a containment mapping from Q to Q' (line 16). This is done by simply checking if the removed subgoal g may be mapped to any subgoal in Q'. If no such mapping exists, then we return transformed rule Q', otherwise `none`.

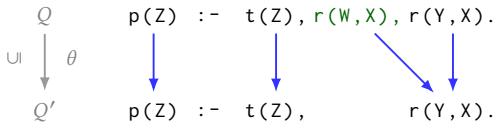
The algorithm for checking the existence of a containment mapping from Q to Q', where Q' is derived from Q by removing a subgoal g is shown in procedure ExistsContainment of Alg. 3. As a first step, we extract all variables *vars* in Q and *vars*′ in Q' (lines 2–3). We then compute the set of removed variables *rmVars* (line 4). Function ReplaceWithWildcard (line 5) creates a pattern expression p from g such that the first occurrence of each variable in g is replaced with a wildcard if the variable is also in *rmVars*. Any subsequent occurrences of the same variable are replaced with a back-reference to the first match; this ensures that equality constraints between variables are captured. In the example of Fig. 11, *rmVars* is empty, so p is g, that is, `r(X,Y)`. If any g' in the body of Q' matches this pattern, then g may be mapped to a subgoal in Q' and a mapping exists (lines 7–9). Otherwise, a mapping does not exist (line 10) as in Fig. 11.

Algorithm 3: REM transformations

```
1  procedure ExistsContainment(Q, Q', g)
2    vars ← ExtractAllVars(Q)
3    vars' ← ExtractAllVars(Q')
4    rmVars ← vars \ vars'
5    p ← ReplaceWithWildcard(g, rmVars)
6    head', body' ← Q'
7    for each g' ∈ body' do
8      if g' matches p then
9        return true
10   return false
11 procedure RemExp(Q)
12   head, body ← Q
13   g ← RandSubgoal(body)
14   Q'.head ← head
15   Q'.body ← body \ {g}
16   if ¬ ExistsContainment(Q, Q', g) then
17     return Q'
18   return none
```



(a) Containment mapping θ from Q to Q'.



(b) Mapping of head and subgoals induced by θ.

Figure 12: Example of REMEQU transformation.

REMEQU. If, after removing a subgoal of Q, a containment mapping θ from Q to Q' does exist, then we have a REMEQU transformation because both $Q' \subseteq Q$ and $Q \subseteq Q'$ hold. The former holds due to θ and the latter due to σ.

Example. Fig. 12 shows an example of a REMEQU transformation.

Algorithm. This algorithm is analogous to the one for REMEXP (see Alg. 3). For this transformation however, we return Q' when a containment mapping from Q to Q' *does* exist, i.e., EXISTSCONTAINMENT on line 16 is not negated. In the example of Fig. 12, *rmVars* is a singleton containing variable `W`, thus pattern p on line 5 of Alg. 3 is `r(*,X)`, where `*` is a wildcard. Subgoal `r(Y,X)` in Q' matches this pattern, and as a result, a mapping exists.

REMCON. Analogously to ADDEXP, this transformation can only be the same as REMEQU.

4.4 Transformation Sequences

Until now, we have focused on applying a single transformation to a single rule, which is a conjunctive query. However, our approach is also able to apply sequences of transformations to such a rule.

More specifically, a rule macro-transformation T may be composed of a sequence of micro-transformations $[t_1, \ldots, t_n]$ as the ones that we described so far. However, every micro-transformation $t_i \in T$ must preserve the intended oracle for the rule (i.e., EXP, EQU, CON). In particular, for an expanding macro-transformation T_{EXP}, in which $Q \subseteq Q'$, the sequence of micro-transformations may have oracles EQU or EXP, but not CON. Analogously, for a contracting macro-transformation T_{CON}, in which $Q' \subseteq Q$, the micro-transformations may have oracles EQU or CON, but not EXP. For an equivalent macro-transformation T_{EQU}, in which $Q \equiv Q'$, all micro-transformations must also have EQU oracles.

In the following section, we generalize our approach from a single conjunctive query to a Datalog program, containing rules that are not necessarily CQs.

5 BEYOND CONJUNCTIVE QUERIES

Let us first show how the oracle of a rule (macro-)transformation generalizes to any *positive*-Datalog program, i.e., any program without negation. To do this, we need to explain *monotonicity* of conjunctive queries. Intuitively, when adding more entries to a database instance, a monotonic query never produces a smaller result.

DEFINITION 5 (MONOTONICITY). *A conjunctive query Q over a database schema $\mathcal{R}$ is monotonic iff, for every two instances I and J of $\mathcal{R}$, it holds that $Q(I) \subseteq Q(J)$ when $I \subseteq J$.*

In a program P, the output relation is called a *Datalog query*, Q_P. Suppose our approach transforms a rule Q in P to get new rule Q', and therefore, new program P' and Datalog query Q'_P. Now, the same oracle that should hold between Q and Q' should also hold between Q_P and Q'_P. This is because, in positive Datalog, all rules are monotonic. Therefore, due to the fixed-point computation, any change in the result of Q propagates monotonically to all rules that (directly or transitively) depend on Q. Ultimately, this includes the final Datalog query, and thus, the program result. Consequently, we may "lift" our oracles from individual conjunctive queries to full-blown positive-Datalog programs. Naturally, this also allows us to transform more than one rule in a positive-Datalog program as long as all transformations have the same intended oracle.

Let us now explain how our approach handles *any* Datalog program (not only positive ones). Of course, the EQU oracle trivially extends to any program. However, queryFuzz is able to accept any Datalog program for *all* oracles: it enforces that all rules depending on a transformed rule Q' are monotonic (e.g., they do not contain any negated subgoals). Intuitively, should the result of a rule Q' "flow" into a non-monotonic rule, the effect on the program result could be "flipped", for instance, it could be contracted instead of expanded. This is undesirable as it could lead to false positives. To handle negation, existing Datalog engines impose a computation order on relations. More specifically, relations are assigned to *strata* via a process known as *stratification* [4, 58]. Lower strata are computed before higher ones during the fixed-point computation. Therefore, queryFuzz works on any Datalog program by only transforming rules that are in a higher stratum than any rules containing negation. As a consequence, no results of transformed rules can "flow" into non-monotonic rules.

Note that many Datalog dialects support rules with more expressive language features, such as comparison operators, aggregate functions, disjunctions, and recursion. While our transformations target the restricted subset of pure conjunctive queries (see Sect. 3),

they may also be applied to more expressive dialects as long as the monotonicity constraints described above are maintained. In fact, our implementation does handle such dialects.

Based on the above, in the rest of this section, we present another transformation in queryFuzz, which is specific to Datalog programs (unlike the transformations in Sect. 4, which target CQs in general).

5.1 Neg Transformation

A Neg transformation changes a program P into an equivalent but further stratified program P' by introducing a double negation in a rule Q. In particular, introducing a negation causes the Datalog engine to split a stratum in two. When this negation is double, we guarantee the Equ oracle (i.e., the transformation is NegEqu).

We introduce so-called *safe* negations, i.e., every variable in a negated subgoal must also appear in a positive subgoal. (Unsafe rules are traditionally not allowed in Datalog as they do not restrict all variables to finite domains.) As an example, consider:

```
p(X,Y) :- a(X,Y), b(Y,Z), c(Z).    // Q
```

NegEqu selects a subgoal g in Q, say `c(Z)`, and replaces it with a new negated subgoal, say `!neg(Z)`. Relation `neg` is defined to have the same body as Q but with a negated g, thus introducing a double negation:

```
neg(Z) :- a(X,Y), b(Y,Z), !c(Z).
p(X,Y) :- a(X,Y), b(Y,Z), !neg(Z). // Q'
```

One can easily see that queries Q and Q' are equivalent when thinking about the transformation logically: $\mathtt{a} \wedge \mathtt{b} \wedge \neg\, \mathtt{neg} \equiv \mathtt{a} \wedge \mathtt{b} \wedge (\neg\, \mathtt{a} \vee \neg\, \mathtt{b} \vee \mathtt{c}) \equiv \mathtt{a} \wedge \mathtt{b} \wedge \mathtt{c}$. Such a transformation partitions the original stratum of relation `p` into two, where the stratum of `p` is strictly greater than that of `c`. Note that Datalog traditionally disallows Neg transformations when g (in this case `c`) has a cyclic dependency on the head of Q (in this case `p`), which would require them to be defined in the same stratum.

6 IMPLEMENTATION

We implemented queryFuzz in a total of 5,300 lines of Python. It currently supports three Datalog dialects, namely Soufflé [32], bddbddb [73] (used by μZ), and DDlog [59]. In the rest of this section, we discuss how we implement the bug-detection stage of our approach.

Bug detection. During bug detection, queryFuzz compares the result of a program (`gen` in Fig. 3) with that of its transformation (`exp`, `equ`, or `con` in the figure). However, a program result could potentially contain millions of entries. This is especially true for randomly generated programs. To efficiently check an oracle, queryFuzz uses Datalog rules that decide result containment.

For instance, the rules that check Equ oracles are the following:

```
equ1(Z) :- gen(Z), !equ(Z).
equ2(Z) :- equ(Z), !gen(Z).
```

The first rule checks whether $\mathtt{gen} \subseteq \mathtt{equ}$ and the second whether $\mathtt{equ} \subseteq \mathtt{gen}$. A bug is detected if the result of either `equ1` or `equ2` is non-empty.

7 EXPERIMENTAL EVALUATION

In this section, we address the following research questions:

Table 1: Query bugs detected by queryFuzz.

Bug ID	Datalog Engine	Metamorphic Transformations	Bug Status
1	Soufflé	Add	fixed
2	Soufflé	Rem, Rem, Rem, Mod	fixed
3	Soufflé	Mod, Add, Add	confirmed
4	Soufflé	Neg	fixed
5	Soufflé	Mod, Add, Add	confirmed
6	Soufflé	Mod, Add, Mod, Rem	confirmed
7	Soufflé	Rem, Mod, Add	confirmed
8	Soufflé	Add, Add, Mod	confirmed
9	μZ	Add, Mod, Add	fixed
10	μZ	Add, Add, Add, Mod	fixed
11	μZ	Add, Mod	fixed
12	μZ	Mod, Add	confirmed
13	DDlog	Add, Add, Add	confirmed

RQ1: How effective is queryFuzz in detecting previously unknown query bugs in Datalog engines?
RQ2: Is the number of detected bugs significant?
RQ3: How deep are the detected bugs?
RQ4: What are characteristics of the detected bugs?
RQ5: How efficient is queryFuzz?

7.1 Experimental Setup

We tested Soufflé, μZ, and DDlog, three popular and mature Datalog engines that are publicly available on GitHub. We completed the development of the first version of queryFuzz, with a subset of the metamorphic transformations and limited support for different language features, in May 2020, and initially focused on testing Soufflé. We only added support for the dialects of μZ and DDlog in late Dec 2020 to evaluate the generality of our transformations.

To avoid burdening developers and reporting duplicate issues, we only filed reports for bugs that were clearly different than the ones we had already reported until these were fixed. Of course, this hinders bug reporting, but it was greatly appreciated by the developers. In fact, we are now closely working with the Soufflé team on integrating queryFuzz in their development cycle.

7.2 Experimental Results

We now discuss our experimental results for each of the above research questions.

RQ1: Query bugs. Tab. 1 shows the list of *unique* query bugs detected by queryFuzz in the Datalog engines we tested. Note that we confirmed bug uniqueness with the engine developers themselves. The first column of the table assigns an identifier to each bug; all identifiers link to the corresponding bug reports on the GitHub issue tracker of each engine. The second column of the table shows the engine in which the bug was found, the third the sequence of metamorphic transformations that revealed the bug, and the last column shows the current status of the bug (i.e., open, confirmed, or fixed).

Overall, queryFuzz detected 13 previously unknown query bugs in all three engines. All bugs have been confirmed by the developers, and 6 have already been fixed. Bugs 3 and 5 are labeled as

Table 2: By-product bugs detected by queryFuzz.

Bug ID	Datalog Engine	Bug Type	Bug Status
14	Soufflé	floating-point exception	confirmed
15	Soufflé	aborted evaluation	fixed
16	Soufflé	segmentation fault	fixed
17	Soufflé	segmentation fault	fixed
18	Soufflé	segmentation fault	fixed
19	Soufflé	segmentation fault	fixed
20	Soufflé	segmentation fault	confirmed
21	Soufflé	assertion failure	fixed
22	Soufflé	assertion failure	fixed
23	Soufflé	assertion failure	confirmed
24	Soufflé	assertion failure	confirmed
25	Soufflé	assertion failure	open
26	Soufflé	compiler error	fixed
27	μZ	performance bug	fixed

questions on the issue tracker even though developers have confirmed them. The reason is that they reveal a deep semantic issue in logic programming that cannot be easily addressed (see RQ4). As shown in the third column of the table, each of our metamorphic transformations (i.e., Add, Mod, Rem, and Neg) contributed to detecting at least one query bug. Moreover, the fact that each tested engine implements its own Datalog dialect speaks to the generality of these transformations. Note, however, that our public bug reports do not all show the applied transformations as we tried to localize issues as much as possible and aid developers in debugging; our tool repository[4] contains instructions on how to reproduce all bug-revealing transformations.

In addition to query bugs, queryFuzz also detected several crash bugs as a by-product; they are shown in Tab. 2. Even though such bugs are less critical, they expose robustness issues, and developers were still interested in them. In fact, a developer of Soufflé said: "*Bug reports like this are definitely welcome, especially because they might also point to other potential issues in our setup. [These issues] have already been super useful.*"

In general, we found many more bugs in Soufflé in comparison to the other engines. However, this does not necessarily mean that Soufflé is more buggy. A reason is that we tested it for a longer period of time (see Sect. 7.1). Another reason is that the Z3 developers currently have very limited bandwidth to devote to μZ as they are working on a new core SMT engine—we, therefore, decided against filing more bugs for the time being. In addition, DDlog is quite slow as it compiles every input program into a Rust project; this also slows down the testing process (see RQ5).

RQ2: Significance of bug numbers. To evaluate the significance of our bug-finding results, we compare the number of query bugs detected by queryFuzz to the total number of such bugs reported from May 1, 2020 to Feb 15, 2021. For this research question, we consider all three engines, and we collect the total number of reported bugs from their GitHub issue trackers. We inspect issues since May 1, 2020 because this is when we started testing Soufflé.

The results are shown in Fig. 13. In the considered time period, a total of 16 query bugs were reported in the three Datalog engines we

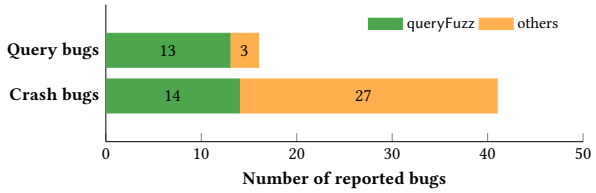


Figure 13: All bugs reported in the three Datalog engines from May 1, 2020 to Feb 15, 2021.

Table 3: Categorization of Soufflé bugs into the components in which they were found.

Soufflé Component	Bug IDs
AstGen	A, E
AstOpt	1, 2, 3, 4, 5, 15, 17, 20, 21, 25, B, C, D
AstRam	8, 22, 23, 24
RamOpt	19
IntSyn	6, 7, 14, 26
Infra	16, 18

tested, and queryFuzz detected 13 of them (81%). This ratio, though very high, is not surprising since query bugs are very hard to detect without an oracle. In the same time period, 41 crash bugs were reported, and queryFuzz detected 14 of them (34%) as a by-product.

RQ3: Bug depth. To understand the depth of the detected bugs, we analyzed all Soufflé bugs together with the engine developers. In general, they revealed issues across the stack.

The Soufflé engine essentially consists of the following components, from front- to back-end: (1) AstGen for parsing and abstract-syntax-tree (AST) generation, (2) AstOpt for AST analysis and optimization, (3) AstRam for translation from AST to relational-algebra machine (RAM), (4) RamOpt for RAM optimization, and (5) IntSyn for interpretation or synthesis. The interpreter evaluates its RAM input, whereas the synthesizer translates RAM into C++ code, which is then compiled and executed.

Tab. 3 categorizes all Soufflé bugs into the engine component in which they were found—ignore bugs A, B, C, D, and E for now. Note that no bugs were found in AstGen and that we include a row Infra, referring to infrastructure bugs, e.g., in utilities, that could affect the entire stack. As shown in the table, queryFuzz detected bugs in all components except AstGen, which is the most shallow.

We compare the depth of these bugs with that of bugs detected using of-the-shelf fuzzers and reported from May 1, 2020 to Feb 15, 2020. There were 6 such bugs, 3 of which were detected with Radamsa [1] and the other 3 with AFL [2]. One of the Radamsa bugs[5] was not confirmed by the developers, who labeled it as 'wont-fix'. The other 5 bugs are shown in Tab. 3 as A, B, C, D, and E. They revealed issues in the AstGen and AstOpt components of Soufflé, which are at the top of the stack—for instance, bugs A and E crash the engine during, or even before, parsing. The reason why queryFuzz bugs are much deeper is that it generates valid Datalog programs and its oracle-driven transformations are more likely to reveal semantic issues. Note that we do not further compare our

[4] https://github.com/Practical-Formal-Methods/queryFuzz

[5] https://github.com/souffle-lang/souffle/issues/1757

```
PQRI(v) :- Z(v), Z(nbj).
PLEY(o) :- PQRI(x), Z(o), Z(x).
NFUV(q) :- Z(fym), PLEY(q).
OUT(t) :- NFUV(ssz), PLEY(arv), PLEY(t).
```

Figure 14: Generated program snippet for testing DDlog.

approach with other off-the-shelf fuzzers as they are not able to detect query bugs due to lack of oracles.

RQ4: Bug characteristics. To demonstrate the nature of the detected bugs, we now provide a few interesting bug samples.

Bug 1 was found in Soufflé's AstOpt component, and specifically, in the minimization pass that simplifies the program by removing equivalent rules and subgoals. This pass missed a corner case for singleton relations, i.e., with arity 1. The program and transformation that revealed this bug are discussed in Sect. 2 (see Fig. 5). Bug 4 was detected in the same component, but in its magic-set transformation [4, 5, 7, 58], which aims to derive only those facts that are relevant for the program's Datalog query. Our approach revealed this bug using a Neg transformation. The rule in which the negation was introduced depended on another rule containing a comparison operator, which in turn caused a mislabeling of relations as positive. Naturally, correct positive labeling is essential to the stratification process. Bug 2 revealed another issue in the magic-set transformation. In general, developers mentioned that implementing optimization passes on the AST is quite complex for a feature-rich Datalog dialect. They also expressed the need for verifying the correctness of such passes, as done by Bégay *et al.* [8].

According to developers, bugs 3 and 5 reveal an important semantic issue in logic programming. There is no clear execution order of instructions, which may result in numerical-stability issues in the presence of floating-point numbers. For these bugs to be fixed, the developers would have to build symbolic machinery that dictates the order of optimizations and instructions such that numerical stability is maximized. However, this is an open research problem, which is why these bugs were labeled as questions.

Bug 7 was detected in Soufflé's IntSyn component, at the very bottom of the stack. According to the developers, the problem lies in a data-structure representation for relations, namely brie [33], which does not properly implement element insertion and count. This bug existed at least since an old release of Soufflé (of 1.5 years ago). A developer commented about this bug: "*I don't know how it could have been missed until now, but that's the first time I've seen anyone point this out.*" Bug 6 revealed a different issue with the same data structure; in this case, the computation of lower and upper bound values of its elements was incorrect.

Bug 8 was found in the AstRam component of Soufflé. Our transformation caused a silent internal failure in this component, which manifested itself with an incorrect result. A developer commented: "*Well spotted! Great work!*"

Bug 13, was detected in DDlog after randomly generating the program in Fig. 14 and then adding two subgoals to rule `PLEY`:

```
PLEY(o) :- PQRI(x), Z(o), Z(x), PQRI(z), PQRI(x).
```

The original program repeatedly computes Cartesian products of the different relations and generates a non-empty result. However, after the above transformation, which should preserve the original result, the new program generates an empty result. This is because DDlog stores all intermediate relations as multi-sets, where the multiplicity of each element is the number of times it was derived. Currently, multiplicities are stored as 32-bit integers to reduce the memory footprint of the program, and the above transformation caused an integer overflow, manifesting itself as an empty result. This bug was confirmed by the developers, who are considering several solutions to the problem, such as using 64-bit integers to store multiplicities, internally converting multi-sets to sets using the `distinct` operator in Rust, or statically analyzing the program to estimate the number of derivations.

RQ5: Performance. Regarding the performance of queryFuzz, it expectedly varies significantly depending on the tested Datalog engine. On average, Soufflé requires 0.078 seconds to run a test (12.9 tests per second) in interpreter mode and 12 seconds in synthesizer mode, DDlog needs 1.2 minutes per test, and μZ 0.1 seconds (10 tests per second). On average, the first stage of queryFuzz generates 47.6 programs per second, and the second stage performs 303 transformations per second. As shown from these numbers, the performance bottleneck are the engines themselves.

7.3 Threats to Validity

We identified two threats to the validity of our experiments.

Selection of seeds. Our approach may use seeds as input, and its effectiveness in bug finding could depend on their selection. However, we used non-empty seeds only when testing Soufflé, and we selected all of its semantically valid regression tests[6]. Our seed selection is, therefore, sufficiently broad to mitigate this threat. Moreover, queryFuzz does not require non-empty seeds as, in their absence, it generates random Datalog programs (see Sect. 2). In fact, 7 of the detected bugs were found using non-empty seeds.

Selection of Datalog engines. The detected bugs also depend on our selection of Datalog engines. However, we chose three mature engines, which even support different dialects, to mitigate this threat and demonstrate the generality of our approach.

8 RELATED WORK

In this paper, we present the first testing approach for detecting query bugs in Datalog engines. It uses metamorphic testing to solve the common problem of finding a suitable oracle [6] taking inspiration from query optimization in database theory. Of course, query optimization has been studied in other domains as well, such as in Datalog or Prolog (e.g., [25, 54, 60, 72]). However, optimization targets a goal different than ours, that of finding an equivalent query that performs faster. In contrast, queryFuzz tests Datalog engines by exploring a state space of queries that are not necessarily equivalent, let alone more optimal. In the following, we focus on testing work from related areas, such as database systems, compilers, and program analyzers.

Metamorphic testing. Metamorphic testing [21] is an effective technique to test software systems without user-provided oracles. It works by mutating test cases via metamorphic relations that allow inferring the expected output of the mutated test cases. Over the years, it has been used to test a variety of systems, from web

[6]We selected all tests in the 'evaluation', 'example', and 'semantic' folders under https://github.com/souffle-lang/souffle/tree/master/tests.

services [18], over compilers [41], to machine-learning applications [77]. Segura *et al.* [62] conducted a comprehensive survey on metamorphic testing in different domains.

Testing database systems. Database-management systems lie at the heart of most large-scale software applications today. Ensuring their correctness and robustness is of critical importance and has been a focus of many researchers and practitioners for decades.

In 1998, Slutz [65] proposed a technique, based on differential testing, to detect bugs in database systems. Another approach—also based on differential testing—was used by Jinho *et al.* to detect performance bugs [34]. Jepsen [39], developed by Kingsbury, is a practical tool for detecting safety bugs in distributed database systems; these can occur due to asynchronous interactions between components, data loss due to networking issues, node failures, etc. Recently, Rigger and Su proposed a series of testing techniques [55–57], which they implemented in a tool called SQLancer. Their tool detected hundreds of bugs in various relational database systems.

Fuzzing is also applied to detect crashes and other robustness issues in database systems. For instance, SQLsmith [63] is a popular SQL-query generator that has detected hundreds of crashes in widely used database systems. Other query-generation approaches include ones relying on constraint solvers [14, 36, 37, 49, 52, 71], symbolic execution [10, 45], and reverse query processing [9].

Testing compilers. Compiler testing is another important and active research area [20, 43, 66, 78]. Le *et al.* proposed a metamorphic-testing technique [41], known as equivalence modulo inputs (EMI), which mutates a seed program to generate equivalent programs. The technique and its extensions [42, 67, 80] have detected hundreds of bugs in GCC and Clang. A related approach was also used to test graphics shader compilers [24, 43]. Livinskii *et al.* recently developed a technique for generating expressive programs without undefined behavior to test C and C++ compilers [44]. The programs are then compiled using different compilers, and their outputs are compared to detect bugs.

Testing program analyzers. Work on detecting bugs—in particular soundness bugs—in implementations of program-analysis techniques [17] has received significant attention in recent years. Various different approaches have been proposed to test a wide range of analysis techniques, such as model checking [79], abstract interpretation [40], symbolic execution [35], or dataflow analysis [68], as well as their underlying components, such as abstract domains [16] or constraint solvers [15, 47, 70, 75, 76].

9 CONCLUSION

We have presented the first approach for metamorphic testing of Datalog engines. Our tool, queryFuzz, detected 13 previously unknown query bugs in three different engines. Query bugs are critical since, unlike crashes, they typically remain undetected. Given that Datalog is frequently used to formalize and implement security analyses or verification tools, such bugs can be catastrophic. As a result, we received overwhelmingly positive reactions from engine developers about the bugs we reported, several of which revealed deep—sometimes even fundamental—issues.

ACKNOWLEDGMENTS

We thank the reviewers for their constructive feedback. We are grateful to the Datalog-engine developers for their valuable help and feedback, and especially to Bernhard Scholz and the Soufflé team. This work was supported by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science).

REFERENCES

[1] [n.d.]. Radamsa. https://gitlab.com/akihe/radamsa.
[2] [n.d.]. Technical "Whitepaper" for AFL. http://lcamtuf.coredump.cx/afl/technical_details.txt.
[3] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *SIGMOD*. ACM, 1371–1382.
[4] Isaac Balbin, Graeme S. Port, Kotagiri Ramamohanarao, and Krishnamurthy Meenakshi. 1991. Efficient Bottom-Up Computation of Queries on Stratified Databases. *JLP* 11 (1991), 295–344. Issue 3&4.
[5] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. 1986. Magic Sets and Other Strange Ways to Implement Logic Programs. In *PODS*. ACM, 1–15.
[6] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *TSE* 41 (2015), 507–525. Issue 5.
[7] Catriel Beeri and Raghu Ramakrishnan. 1991. On the Power of Magic. *JLP* 10 (1991), 255–299. Issue 3&4.
[8] Pierre-Léo Bégay, Pierre Crégut, and Jean-François Monin. 2021. Developing and Certifying Datalog Optimizations in Coq/MathComp. In *CPP*. ACM, 163–177.
[9] Carsten Binnig, Donald Kossmann, and Eric Lo. 2007. Reverse Query Processing. In *ICDE*. IEEE Computer Society, 506–515.
[10] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. 2007. QAGen: Generating Query-Aware Test Databases. In *SIGMOD*. ACM, 341–352.
[11] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-To Analyses. In *OOPSLA*. ACM, 243–262.
[12] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *CoRR* abs/1809.03981 (2018).
[13] Neville Brent, Lexiand Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. In *PLDI*. ACM, 454–469.
[14] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. 2006. Generating Queries with Cardinality Constraints for DBMS Testing. *TKDE* 18 (2006), 1721–1725. Issue 12.
[15] Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *ICSE*. ACM, 1459–1470.
[16] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically Testing Implementations of Numerical Abstract Domains. In *ASE*. ACM, 768–778.
[17] Cristian Cadar and Alastair F. Donaldson. 2016. Analysing the Program Analyser. In *ICSE*. ACM, 765–768.
[18] W. K. Chan, S. C. Cheung, and Karl R. P. H. Leung. 2005. Towards a Metamorphic Testing Methodology for Service-Oriented Software Applications. In *QSIC*. IEEE Computer Society, 470–476.
[19] Ashok K. Chandra and Philip M. Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *STOC*. ACM, 77–90.
[20] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surv.* 53 (2020), 4:1–4:36. Issue 1.
[21] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases.* Technical Report HKUST–CS98–01. HKUST.
[22] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. 2007. .QL: Object-Oriented Queries Made Easy. In *GTTSE (LNCS, Vol. 5235)*. Springer, 78–133.
[23] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340.
[24] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *PACMPL* 1 (2017), 93:1–93:29. Issue OOPSLA.
[25] Markian M. Gooley and Benjamin W. Wah. 1988. Efficient Reordering of Prolog Programs. In *ICDE*. IEEE Computer Society, 110–117.
[26] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing Software Verifiers from Proof Rules. In *PLDI*. ACM, 405–416.

[27] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *PACMPL* 2 (2018), 116:1–116:27. Issue OOPSLA.

[28] Sergio Greco and Cristian Molinaro. 2015. *Datalog and Logic Databases*. Morgan & Claypool.

[29] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. 2014. Demonstration of the Myria Big Data Management Service. In *SIGMOD*. ACM, 881–884.

[30] Krystof Hoder, Nikolaj Bjørner, and Leonardo de Moura. 2011. μZ—An Efficient Engine for Fixed Points with Constraints. In *CAV (LNCS, Vol. 6806)*. Springer, 457–462.

[31] Seo Jiwon, Guo Stephen, and Lam Monica S. 2013. SociaLite: Datalog Extensions for Efficient Social Network Analysis. In *ICDE*. IEEE Computer Society, 278–289.

[32] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On Synthesis of Program Analyzers. In *CAV (LNCS, Vol. 9780)*. Springer, 422–430.

[33] Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019. Brie: A Specialized Trie for Concurrent Datalog. In *PPoPP*. ACM, 31–40.

[34] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woon-Hak Kang. 2019. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *VLDB* 13 (2019), 57–70. Issue 1.

[35] Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *ASE*. IEEE Computer Society, 590–600.

[36] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O. Laleye, and Sarfraz Khurshid. 2008. Query-Aware Test Generation Using a Relational Constraint Solver. In *ASE*. IEEE Computer Society, 238–247.

[37] Shadi Abdul Khalek and Sarfraz Khurshid. 2010. Automated SQL Query Generation for Systematic Testing of Database Engines. In *ASE*. ACM, 329–332.

[38] Ross D. King. 2004. Applying Inductive Logic Programming to Predicting Gene Function. *AI Mag.* 25 (2004), 57–68. Issue 1.

[39] Kyle Kingsbury. [n.d.]. Jepsen. https://jepsen.io.

[40] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially Testing Soundness and Precision of Program Analyzers. In *ISSTA*. ACM, 239–250.

[41] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *PLDI*. ACM, 216–226.

[42] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *OOPSLA*. ACM, 386–399.

[43] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *PLDI*. ACM, 65–76.

[44] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *PACMPL* 4 (2020), 196:1–196:25. Issue OOPSLA.

[45] Eric Lo, Carsten Binnig, Donald Kossmann, M. Tamer Özsu, and Wing-Kai Hon. 2010. A Framework for Testing DBMS Features. *VLDB* 19 (2010), 203–230. Issue 2.

[46] Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2009. Declarative Networking. *CACM* 52 (2009), 87–95. Issue 11.

[47] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In *ESEC/FSE*. ACM, 701–712.

[48] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10 (1998), 100–107. Issue 1.

[49] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. 2008. Generating Targeted Queries for Database Testing. In *SIGMOD*. ACM, 499–510.

[50] Raymond J. Mooney. 1996. Inductive Logic Programming for Natural Language Processing. In *ILP (LNCS, Vol. 1314)*. Springer, 3–22.

[51] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *PLDI*. ACM, 308–319.

[52] Meikel Poess and John M. Stephens. 2004. Generating Thousand Benchmark Queries in Seconds. In *VLDB*. Morgan Kaufmann, 1045–1053.

[53] David Poole. 1995. Logic Programming for Robot Control. In *IJCAI*. Morgan Kaufmann, 150–157.

[54] Raghu Ramakrishnan, Catriel Beeri, and Ravi Krishnamurthy. 1988. Optimizing Existential Datalog Queries. In *PODS*. ACM, 89–102.

[55] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *ESEC/FSE*. ACM, 1140–1152.

[56] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *PACMPL* 4 (2020), 211:1–211:30. Issue OOPSLA.

[57] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *OSDI*. USENIX, 667–682.

[58] Kenneth A. Ross. 1990. Modular Stratification and Magic Sets for Datalog Programs with Negation. In *PODS*. ACM, 161–171.

[59] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog (CEUR, Vol. 2368)*. CEUR-WS.org, 56–67.

[60] Yehoshua Sagiv. 1988. Optimizing Datalog Programs. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 659–698.

[61] José Carlos Almeida Santos, Houssam Nassif, David Page, Stephen H. Muggleton, and Michael J. E. Sternberg. 2012. Automated Identification of Protein-Ligand Interaction Features Using Inductive Logic Programming: A Hexose Binding Case Study. *BMC Bioinform.* 13 (2012), 162.

[62] Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *TSE* 42 (2016), 805–824. Issue 9.

[63] Andreas Seltenreich. [n.d.]. SQLsmith. https://github.com/anse1/sqlsmith.

[64] Alexander Shkapsky, Kai Zeng, and Carlo Zaniolo. 2013. Graph Queries in a Next-Generation Datalog System. *VLDB* 6 (2013), 1258–1261. Issue 12.

[65] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *VLDB*. Morgan Kaufmann, 618–622.

[66] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and Analyzing Compiler Warning Defects. In *ICSE*. ACM, 203–213.

[67] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *OOPSLA*. ACM, 849–863.

[68] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing Static Analyses for Precision and Soundness. In *CGO*. ACM, 81–93.

[69] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *CCS*. ACM, 67–82.

[70] Muhammad Usman, Wenxi Wang, and Sarfraz Khurshid. 2020. TestMC: Testing Model Counters Using Differential and Metamorphic Testing. In *ASE*. IEEE Computer Society, 709–721.

[71] Manasi Vartak, Venkatesh Raghavan, and Elke A. Rundensteiner. 2010. QRelX: Generating Meaningful Queries that Provide Cardinality Assurance. In *SIGMOD*. ACM, 1215–1218.

[72] Jian Wang, Jungsoon P. Yoo, and Thomas J. Cheatham. 1993. Efficient Reordering of C-PROLOG. In *Conference on Computer Science*. ACM, 151–155.

[73] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *APLAS (LNCS, Vol. 3780)*. Springer, 97–118.

[74] John Whaley and Monica S. Lam. 2004. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI*. ACM, 131–144.

[75] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *PACMPL* 4 (2020), 193:1–193:25. Issue OOPSLA.

[76] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *PLDI*. ACM, 718–730.

[77] Xiaoyuan Xie, Joshua Wing Kei Ho, Christian Murphy, Gail E. Kaiser, Baowen Xu, and Tsong Yueh Chen. 2009. Application of Metamorphic Testing to Supervised Classifiers. In *QSIC*. IEEE Computer Society, 135–144.

[78] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*. ACM, 283–294.

[79] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and Understanding Bugs in Software Model Checkers. In *ESEC/FSE*. ACM, 763–773.

[80] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *PLDI*. ACM, 347–361.