D4: Debugging Databases During Development

Alperen Keles
University of Maryland, College Park
College Park, USA
akeles@umd.edu

Ethan Chou
University of Maryland, College Park
College Park, USA
echou1@umd.edu

Harrison Goldstein
University of Maryland, College Park
College Park, USA
harry@umd.edu

Leonidas Lampropoulos
University of Maryland, College Park
College Park, USA
leonidas@umd.edu

ABSTRACT

Database management systems (DBMSs) are notoriously complex, making them difficult to test effectively, especially during early development when many features are incomplete. Traditional testing tools like SQLancer and SQLSmith are highly effective for mature databases, but they struggle with high false positive rates and low actionability when applied to evolving systems.

We present D4, a paradigm designed specifically for debugging databases during development, which integrates a testing framework directly into the DBMS, enabling the random testing process to evolve in tandem with the system and reducing false positives by construction. We introduce generation actions, an abstraction for allowing database developers rather than testing experts to specify correctness properties. Finally, we develop a novel human-in-the-loop counterexample minimization approach to produce actionable bug reports.

We evaluate D4 on SomeDB[1], an actively developed SQLite-compatible OLTP engine, and show that it finds 23 unique, confirmed bugs–significantly outperforming off-the-shelf SQLancer variants in terms of true positive rate and usefulness of bug reports. Our results demonstrate that embedding testing infrastructure within the DBMS can dramatically improve its effectiveness and usability during development.

ACM Reference Format:
Alperen Keles, Ethan Chou, Harrison Goldstein, and Leonidas Lampropoulos. 2025. D4: Debugging Databases During Development. In . ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Database Management Systems (DBMSs) are complex systems, and inevitably, such complexity leads to bugs. To discover bugs, DBMSs have historically used random testing. Prior work in this domain has ranged from binary fuzzing with AFL [6] to structured query generation via SQLSmith [23], to finding logic bugs via test oracles in SQLancer [21]. Within prior work, SQLancer is distinguished by using specialized test oracles that reveal logic bugs in DBMSs, and it has proven extremely successful at finding bugs in mature databases.

Unfortunately, off-the-shelf testing frameworks are not as effective at finding actionable bugs in DBMSs that are actively being developed and have many missing features. When the space of randomly generated inputs is larger than the system under test supports, many of the generated tests lie in the unimplemented portion of the system, registering as false positives. Naturally, such false counterexamples are of no help to developers of DBMSs.

One way to alleviate this issue is to tailor the testing framework to the DBMS under test, ensuring that the testing framework only tests implemented features. Unfortunately, existing frameworks are not built with this flexibility in mind. In an off-the-shelf tester, such tailoring would require developers of the DBMS to dive into the internals of an external project, modify it to suit their needs, and maintain those changes as the DBMS evolves.

We propose *database-integrated testing*, an alternative testing paradigm for debugging databases during development. We embed the testing tool within the database itself, allowing for the generators to naturally evolve alongside the DBMS, therefore avoiding false positives by construction. At the same time, we leverage this tight integration to empower developers to write their own oracles by developing a domain-specific language (DSL) for expressing properties customized to particular features being developed. We also develop a novel human-in-the-loop counterexample minimization approach that incorporates developer feedback. Our hybrid approach aims for significantly less complex counterexamples than generic methods can achieve in the stateful setting of testing databases, resulting in bug reports that are much more actionable.

We study this problem in the context of SomeDB[1], an open-source SQLite3-compatible OLTP database engine in active development. With an average of more than 30 commits a day, SomeDB is a rapidly evolving target for testing. D4 helped find 23 confirmed bugs, with 19 of them already fixed with the aid of our hybrid minimization approach. In comparison, running SQLancer-SQLite3 with minimal modifications in SomeDB led to a false positive rate of 96.5% with only one new bug, and running SQLancer-SomeDB, an existing independently developed SQLancer integration for SomeDB, without modifications led to a false positive rate of 58% with six newly discovered bugs.

[1]SomeDB is anonymized as part of the double-blind submission process, we will deanonymize SomeDB as well as provide all the artifacts of D4 as open source for the camera-ready.

Conference'17, July 2017, Washington, DC, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In summary, our contributions are as follows:

(1) We propose D4, a paradigm for testing databases during development which integrates a testing framework within the database and provides convenient and flexible abstractions to developers while increasing the rate of true positive bug reports when testing.
(2) As part of D4, we propose a novel interactive shrinking strategy that significantly reduces counterexample complexity by involving developers in the shrinking process.
(3) We implement D4 for SomeDB, an open source SQLite3-compatible OLTP database under active development.
(4) We evaluate the effectiveness of D4 by demonstrating its effectiveness in finding actionable bugs in SomeDB. D4 found 23 unique confirmed bugs with very few false positives.

2 BACKGROUND AND RELATED WORK

We begin by providing relevant background, first on coverage-guided fuzzing and property-based testing at a high level, and then focusing on SQLancer, the most prominent framework for testing databases.

2.1 Coverage-Guided Fuzzing and Property-Based Testing (PBT)

Coverage-guided fuzzing and property-based testing are both established techniques for finding bugs in software. While they approach bug-finding from different perspectives, they are not fundamentally dissimilar.

On one hand, the fuzzing community primarily focuses on smart and effective test-case generation that works out of the box with minimal user input. The key idea is to leverage *runtime feedback*, usually in the form of branch coverage [4, 6], though not always [18], to keep track of inputs that exhibit interesting behavior (e.g. uncovered new paths) and then mutate them in the hopes of discovering even more interesting inputs. The most common oracle for deciding if an input is a bug or not is staggeringly simple: does the program crash? Other oracles include fuzzing against a model or a different implementation (known as differential fuzzing [8, 12]) or using domain-specific feedback such as time to run particular inputs [15].

On the other hand, the PBT community primarily focuses on empowering users to write their own oracles [5, 11]. PBT frameworks usually offer DSLs for expressing oracles in the form of universally quantified executable predicates, as well as infrastructure for writing *generators* - programs that generate test inputs to test such predicates [5, 14].

Although traditionally these communities have been mostly distinct, the lines between them are increasingly blurred. In fact, fuzzing can be viewed as an instance of PBT where the property is fixed (e.g. the program does not crash) and PBT can be improved by extending its generation with coverage-guided capabilities. Such approaches have been tried recently with great success [13, 17], and in the context of database testing, both SQLancer and D4 follow such a hybrid viewpoint.

2.2 SQLancer

SQLancer is the result of a multi-year research project by Rigger et. al that has set the bar for automated random testing of databases since its inception in 2020. It hosts an extensible core with adapters for many production DBMSs including but not limited to ClickHouse, Apache Datafusion, MySQL, PostgreSQL, and SQLite. Within the last five years, SQLancer has not only grown with respect to the breadth of databases it supports, but it has also widened its arsenal of oracles. It started with Pivoted Query Synthesis (PQS) [22], a rather "simple" containment property over databases that has found at least 121 unique logic bugs in production databases. Two metamorphic oracles followed: Non-Optimizing Reference Engine Construction (NoREC) [19], which found 51 optimization bugs, and Ternary Logic Partitioning (TLP) [20], which discovered 77 novel logic bugs.

In addition to PQS, NoREC, and TLP, SQLancer currently supports Query Plan Guidance (QPG) [1] for feedback-guided generation, Cardinality Estimation Restriction Testing (CERT) [2] for finding performance bugs in DBMSs, Differential Query Plans (DQP) [3] for detecting bugs in the join optimizations, and lastly, Constant Optimization Driven Database System Testing (CODDTest) [27] for finding logic bugs using constant folding and propagation techniques. As SQLancer focuses on testing large classes of behaviors across a variety of databases, each oracle amounts to a significant research contribution in a new research paper.

Integrating a new DBMS to SQLancer is a time-consuming but straightforward process if SQLancer hosts a DBMS with similar semantics to the new one. At a minimum, SQLancer integration requires implementing AST connectors, generation APIs for the relevant queries, and the oracles to use for detecting bugs. SQLancer also has a notion of *expected errors*, bugs that the users can deem as expected, so SQLancer does not report them as errors and continues testing.

In terms of counterexample minimization, SQLancer offers two possibilities. The first is a C-Reduce [24] style statement-level minimizer, and the other is an AST-level minimizer that can also minimize individual statements themselves. Reducers implemented as part of the oracles can also be applied to counterexamples.

3 SPECIFYING CORRECTNESS ORACLES

Rather than expecting database developers to modify SQLancer to suit their particular needs, we instead wanted to offer them flexible and extensible abstractions so that they test their database throughout its development. D4 was created to provide such random testing infrastructure that evolves alongside the DBMS and produces actionable bug reports for the database developers. This evolution involves not only tailoring the input space of SQL generation to the currently implemented portion of SomeDB for reducing false positives, but also tailoring the correctness oracles to new features as they are added, decreasing false negatives.

In this section, we will introduce *generation actions*, an imperative formulation for properties that do not just describe *what* the property tests, but *how* to test it. We will begin by using a very simple commutativity property to convey the basic ideas and notation. We will then detail the oracles that were actually developed to test SomeDB, and conclude the section with a description of our generation strategy.

For concreteness, let us consider the following equivalence relation leveraging the commutativity of $(\wedge)$: for any database db and any two predicates p and q that can range over variables from the database, if we evaluate $SELECT\ (p \wedge q)$ and $SELECT(q \wedge p)$ in db they should yield equivalent results:

$$\forall db, p, q. variables(p) \subseteq db \wedge variables(q) \subseteq db$$
$$\implies SELECT(p \wedge q) \equiv SELECT(q \wedge p)$$

Testing such a property would entail generating an arbitrary database, two expressions p and q that only mention variables from that database, and then evaluating it repeatedly.

That is, we have precisely described *what* we want to test, the exact conditions in which the test is valid, but left the *how* to a database-agnostic framework. Instead of quantifying over db, p, q and constraining them with post hoc relations, we propose a simple abstraction for describing *how* db, p, q are generated with respect to the constraints. We call this abstraction for describing properties *Generation Actions (GA)*.

Each generation action is parameterized by the type it returns and the context it can use (implemented as a trait in Rust). As an example, we can express the same high-level commutativity property as a generation action that returns a `property` parameterized by a context that contains a database `db`:

```
gen property db =
  t ⟵ pick db.tables
  c ⟵ pick t.columns
  v ⟵ genOf expression c.type

  p := t.c = v

  q ⟵ gen expression (t,c)

  ! r1 := SELECT (p AND q)
  ! r2 := SELECT (q AND p)
  ! assert(r1 == r2)
```

Rather than independently generating p and q, we can, for the sake of the example, fix some of the details of generation. As shown in the code of the action, we can, for example, `pick` an arbitrary table from the database, `pick` a column from that table, and generate an expression `v` of that column's type, before constructing an explicit equality check `t.c = v`. Alternatively, we could generate an arbitrary `expression` `q` that can refer to both `t` and `c`. We use ⟵ to bind the results of generation and := for regular let-style binding.

We can also express interactions with the database, explicitly annotated with a !. In this example, the only interactions are defining the two different ways of conjuncting p and q and then `assert`ing their equality, but can generally be a query or an assertion.

An interaction not shown in this example that we will use in our oracles is the ability to inject simulated *faults*. DBMSs interact heavily with the underlying OS, file system, and network, all of which have unpredictable and chaotic failure modes. As such, the same set of interactions (e.g. `CREATE`-`INSERT`-`DELETE`) might result in different results depending on any invisible failures in the middle. FoundationDB [28] is heavily praised for its use of simulation testing, injecting simulated faults within an otherwise correctly working testing environment. We also introduced fault injection in D4 as part of the DSL for generation actions, as exemplified by the header initialization bug studied in §5.1.

3.1 Definitions of Oracles in D4

In this subsection, we walk over definitions of six different oracles written as sequences of generation actions, three of which are reimplementations of existing oracles in SQLancer. We start by defining the first of the three SQLancer oracles we implemented, Pivoted Query Synthesis (PQS) [22], as a universally quantified proposition in Fig. 1. PQS states that given a set of tables in the database, `SELECT`ing for a row constructed from the contents of those tables should contain the row.

Fig. 1 shows two formulations of PQS side by side, the propositional formulation with universally quantified variables that define *what* PQS is on the left and a GA formulation of PQS with two tables/columns as generation actions that define *how* PQS is tested on the right, which closely resembles the implementation of PQS that tests SomeDB today.

The propositional formulation quantifies over a database and a sequence of tables, columns, and expressions, under the constraint that the expressions will return $TRUE$ when tested against a row r. The corresponding GA closely follows along, but ensures these constraints are satisfied by construction. We start by picking tables `t1, t2` from the state `db.tables`, followed by picking columns from the respective table. `gen row t` and `gen expression (t, r)` are dependent generation primitives, where the former generates a row based on the table `t`, and the latter generates an expression that will evaluate to `TRUE` for row `r` of the table `t`.

In Fig. 2, we provide definitions of SQLancer oracles as GAs in addition to the other oracles we implemented for D4. In Fig. 2a, we demonstrate how to write Non-Optimizing Reference Engine Construction Generation (NoREC) [19] as a GA. In Fig. 2b, we present the GA for `WHERE Extended` case of Ternary Logic Partitioning (TLP) [20] oracle from SQLancer.

We can express other properties that are not present in SQLancer, including those fundamental to key-value stores such as *Deleted rows should not be in the table* presented in Fig. 2c.

Properties, depending on how they are constructed, might sometimes be invalid. In Fig. 2d, applying `WHERE` or `UNION ALL` operation for composing the results of multiple `WHERE` queries require both sides of the `WHERE`s to have the same number of columns.

We can also write microproperties that reason over very specific sequence of interactions, essentially allowing for creating *regression properties* that apply to situations that previously caused bugs. Fig. 2e presents an example of a microproperty asserting that creating the same table twice causes an error.

In addition to the interaction level properties that are easily extensible, we also provide higher level correctness primitives similar to the SQLancer oracles. As SomeDB is expected to be a drop-in replica of SQLite in the long term, we can use *differential testing*. Another oracle we implemented is for *determinism testing*, which runs the same sequence of interactions for two identical databases and checks if the resulting binaries are identical.

$$
\begin{aligned}
&db : Database\\
&t_1, t_2 ... t_n : Table\\
&c_1, c_2 ... c_n : Column\\
&p_1, p_2 ... p_n : Expression\\
&r : Row\\
&\forall db, t_1, t_2 ... t_n, c_1, c_2 ... c_n, p_1, p_2 ... p_n, r.\\
&t_1, t_2 ... t_n \in db\\
&\ \wedge c_1 \in t_1 \wedge c_2 \in t_2 ... \wedge c_n \in t_n\\
&\ \wedge r.c_1 \in t_1.c_1 \wedge r.c_2 \in t_2.c_2 ... \wedge r.c_n \in t_n.c_n\\
&\ \wedge p_1(r) = TRUE \wedge p_2(r) = TRUE ... \wedge p_n(r) = TRUE\\
&\ \implies r \in SELECT * FROM\ t_1.c_1 ... t_n.c_n\ WHERE\ p_1 ... AND\ p_n
\end{aligned}
$$

```
gen property db:
    t1 ⟵ pick db.tables
    t2 ⟵ pick db.tables

    c1 ⟵ pick t1.columns
    c2 ⟵ pick t2.columns

    r1 ⟵ gen row t1
    r2 ⟵ gen row t2

    r  := (r1.c1, r2.c2)

    ! INSERT INTO t1 VALUES r1
    ! INSERT INTO t2 VALUES r2

    p1 ⟵ gen expression (t1, r1)
    p2 ⟵ gen expression (t2, r2)

    ! RS := SELECT r FROM t1, t2 WHERE p1 AND p2
    ! assert(r in RS)
```

Figure 1: Pivoted Query Synthesis as a universally quantified property (left) and as a generation action (right).

3.2 Query Generation

We designed our query generation algorithm with three goals in mind. First, we wanted each automatically generated database interaction to respect database state (e.g. we should not select from a table that does not exist unless a user-written generation action explicitly calls for it). Second, we did not want to rely on querying the database itself to obtain the information necessary to ensure the first goal—doing so would assume that those queries ran successfully, defeating the purpose of testing. Finally, we wanted to allow database developers to specify, in a lightweight manner, aspects of the distribution of generated interactions that they deemed important, such as read/write heavy queries.

To that end, we follow a generation-by-execution style approach [10], in which a model of the database as a key-value store is updated during generation as a shadow state. Unlike traditional model-based properties [11], this model is not used for differential testing, only for keeping track of the relevant information for correct generation. Algorithm 1 sketches our approach.

Algorithm 1 D4 Generation Algorithm

```
$e \leftarrow \{read : R, write : W, create : C\}$        ▷ Workload distribution
$interactions \leftarrow []$
$st \leftarrow \{read : 0, write : 0, create : 0, tables : []\}$
while $N > 0$ do
    $i \leftarrow gen\ Interaction(st, e)$
    $i.shadow(state)$        ▷ Update the shadow state
    $interactions.push(i)$
    $N \leftarrow N - 1$
end while
```

Database developers can specify three parameters, R, W, and C, which describe the proportion of read, write, and create instructions that should appear in the generated interactions. At each step, the algorithm generates one or more interactions that are compatible with the current shadow state, updates this state to account for the new interactions, and repeats until enough interactions have been generated.

The way we generate individual interactions is shown in Algorithm 1, choosing between generating a single interaction or selecting among the user-defined generation actions to instantiate. We assign a higher generation likelihood to interactions and actions that bias the distribution towards the expected workload target. We show an excerpt of that distribution in Fig. 3.

As mentioned in [22], SQLancer does not keep a separate state, but instead uses the database APIs for querying the current state, such as the table names in `sqlite_master`, due to the implementation effort for the shadow model. We opted for the shadow state because it allows for complex reasoning over the database state for constructing arbitrary queries and properties, and because it gives us a canonical property over the database state: the shadow state is identical to the database at any given moment.

4 SHRINKING STATEFUL SEQUENCES

Counterexample minimization is a fundamental part of random testing. Simply finding bugs is not good enough—a developer must also be able to actually fix the reported bug. A standard approach to perform such minimization is shrinking or delta-debugging [16, 25, 26]: given a randomly generated input $I : T$, a reproducer function $R : T \rightarrow bool$ that determines if a given input reproduces the bug found by I, and a shrinking function $S : T \rightarrow [T]$ that produces smaller versions of the input, shrinking iteratively tries the shrinking function until a smaller input that still triggers the counterexample is found, and repeats the process until reaching a local

```
gen property db:
    t ⟵ pick db.tables
    r ⟵ pick t.rows
    p ⟵ gen expression (t, r)

    ! RS1 = SELECT * FROM t WHERE p
    ! RS2 = SELECT p FROM t
    ! assert(RS1.length() == RS2.count(1))
```

(a) Non-Optimizing Reference Engine Construction (NoREC)

```
gen property db:
    t ⟵ pick db.tables
    p ⟵ gen expression (t)
    p' ⟵ gen expression

    ! RS1 = SELECT * FROM t WHERE p
    ! RS2 = SELECT * FROM t WHERE p AND p' UNION ALL
          SELECT * FROM t WHERE p AND (NOT p') UNION ALL
          SELECT * FROM t WHERE p AND (p' is NULL)
    ! assert(RS1.length() == RS2.count(1))
```

(b) Ternary Logic Partitioning (TLP) WHERE Extended

```
gen property db:
    t ⟵ pick db.tables
    r ⟵ pick t.rows
    p ⟵ gen expression (t, r)

    ! DELETE * FROM t WHERE P
    ! RS = SELECT * FROM t WHERE p
    ! assert(r not in RS)
```

(c) "Deletes rows should not be in the table"

```
gen property db:
    s1 ⟵ gen SELECT db.tables
    s2 ⟵ gen SELECT db.tables

    ! RS3 = s1 UNION ALL s2
    ! assume(s1.first().length() == s1.first().length())
    ! assert(RS1.length() + RS2.length() == RS3.length())
```

(d) "UNION ALL preserves cardinality"

```
gen property:
    t1 ⟵ gen id
    c1 ⟵ gen id
    ct1 ⟵ gen columntype
    c2 ⟵ gen id
    ct2 ⟵ gen columntype

    ! CREATE TABLE t1 (c1 ct1)]
    ! error := CREATE TABLE t1 (c1 ct1)]
    ! assert(error == "Parse error: Table {t1} already exists")
```

(e) "The same table cannot be created twice"

Figure 2: Definitions of Oracles as Generation Actions

$$min(e.read - state.read, e.write - state.write) \mapsto genPQS(st)$$
$$(e.read - state.read)\ /\ 2 \mapsto gen\ NoREC(st)$$
$$(e.read - state.read)\ /\ 2 \mapsto gen\ TLPWhere(st)$$
$$\dots$$
$$(e.read - state.read) \mapsto gen\ SELECT(st)$$
$$(e.write - state.write) \mapsto gen\ INSERT(st)$$
$$(e.write - state.write) \mapsto gen\ UPDATE(st)$$
$$(e.write - state.write) \mapsto gen\ DELETE(st)$$
$$(e.create - state.create) \mapsto gen\ CREATE(st)$$

Figure 3: Weights of selected individual interactions, biasing generation towards the user-specified workload target.

minimum. Algorithm 2 provides the best-first search shrinking loop.

Algorithm 2 $shrink(I : T, R : T \rightarrow bool, S : T \rightarrow [T]) \rightarrow T$

```
$result \leftarrow I$
$ws \leftarrow [I]$
while $ws.len() > 0$ do
    $e \leftarrow ws.pop()$
    if $R(e)$ then
        $ws \leftarrow ws + S(e)$
        $result \leftarrow e$
    end if
end while
```

Shrinking involves a fundamental trade-off depending on the complexity of the shrinking function that generates candidates for shrinking. On one extreme, trying *all* possible smaller inputs would by construction reach a globally minimal counterexample.

However, such an approach would be intractably expensive. On the other extreme, trying too few smaller inputs might be much faster, but risks not actually finding any smaller inputs that still trigger the discovered bug. As a result, shrinking necessarily balances completeness and effectiveness of the minimization with the efficiency of the search.

In the context of database testing, the problem is exacerbated: the input type T is often a sequence of SQL statements, which means the shrinker will have type $S : [SQLStatement] \rightarrow [[SQLStatement]]$, with a total of 2^N potential smaller sequences of statements for a given sequence of length N, independent of the concrete shrinker function. PBT libraries usually solve this exponential blow-up by running incomplete, heuristic-based search algorithms [16]. A standard simplifying assumption [5] is to assume independence between statements, trying to independently remove one statement at a time rather than trying all combinations of them. However, in the stateful world of database testing, such an assumption can only get you so far.

Consider the following sequence of interactions that creates two tables, selects from the second, and inserts NULL into the first:

```
(1) CREATE TABLE t0 (c0);
(2) CREATE TABLE t1 (c1);
(3) SELECT * FROM t1;
(4) INSERT INTO t0 VALUES (NULL);
```

Assume now that an error arises when inserting a NULL value into the table t0, with creating the table at (1) and inserting at (4) sufficient for minimal reproduction. Originally, executing instruction (4) produces a runtime failure, which the reproducer function is looking to replicate. A shrinker that assumes independence between each statement will start by removing (1) and running the reproducer, which results in a different failure F', because the table $t0$ does not exist for the sequence. The shrinker concludes that (1) contributes to the failure and should not be removed. In the next iteration, the shrinker removes (2), this time resulting in another error in (3) because $t1$ does not exist. The shrinker concludes (2) is essential to reproduction, then removes (3) successfully. Thus, (1)-(2)-(4) is reported as the minimal counterexample instead of (1)-(4). The issue is *path dependence*, removing (2) then (3) produces a different result than removing (3) then (2).

The dependencies in a real database execution are much more complex than in this toy example, as the tables reside in the same physical space, so every write to the database potentially affects the results of all statements afterwards.

Another layer of dependencies is an artifact of our stateful generation strategy detailed in §3. Consider the following sequence of interactions resulting from running the generation actions in Figure 1.

```
(1) CREATE TABLE t1 (c1 INT, c2 INT);
-- begin testing 'PQS'
(2) INSERT INTO t1 VALUES (10, 15);
(3) RS = SELECT c1 FROM t1 WHERE c2 > 10;
(4) assert (10) in RS
-- end testing 'PQS'
```

Some embedded assertions rely on specially crafted states of the database. For instance, (4) is only valid if (2) is executed, because the predicate of the WHERE clause in (3) relies on the inserted value in (2). If (2) is removed in the process of shrinking, the assertion in (4) still fails and produces a *false positive*. As a solution, we can hard-code n-tuples of interactions together so that the shrinker will only remove them together, not separately.

Unfortunately, shrinking the counterexample by removing interactions from the generated sequence still results in overly complex counterexamples because individual statements can have arbitrarily complex effects. An INSERT statement may insert 100 randomly generated rows into the database or invoke another randomly generated WHERE, which can WHERE multiple tables or use compound operators. The predicate expressions in the result columns or the where clause of WHERE queries are essentially unbounded arithmetic and logical expressions with bespoke semantics which can trigger bugs in the query optimizer [19]. In corollary, shrinking interaction sequences by removing interactions is required, but not sufficient. We need mechanisms for shrinking the statements themselves too.

4.1 Shrinking in D4

In D4, we have implemented two automated shrinking mechanisms for reducing the length of the generated interaction sequences. The first is a heuristic shrinker, relying on heuristics such as isolation, where any statement referencing a set of tables only affects those tables. The second is a brute-force shrinker that recursively removes interactions, similar to the Algorithm 2 presented at the beginning of this section.

To overcome the practical limitations of these approaches, we developed a hybrid interactive shrinking method, which we call *Human-in-the-Loop shrinking*. We created a textual human-in-the-loop shrinker where developers testing their changes to SomeDB can inspect the generated interaction plan, remove interactions based on their domain-specific knowledge, and tinker with the output. The resulting counterexample can then be fed again to the automatic shrinker, a process which can be repeated to reach a counterexample that is both smaller than automatic methods could achieve on their own, and obtained faster than a human tester could simplify.

In Fig. 4, we depict a short sequence of actions resulting in the successful removal of one interaction from the failing test case produced by random generation. The user starts with a six interaction test case, tries removing the second interaction. Observing the feedback from the shrinker, they revert the removal, remove the third interaction successfully and reduce the sequence to five interactions.

5 EVALUATION

In order to evaluate the effectiveness of D4, we explored the answers to the following questions:

- RQ1: Does D4 find bugs in SomeDB?
- RQ2: How does D4 compare to existing state-of-the-art in terms of bug finding performance?
- RQ3: How much does human-in-the-loop shrinking benefit counter-example minimization on top of automated shrinking?

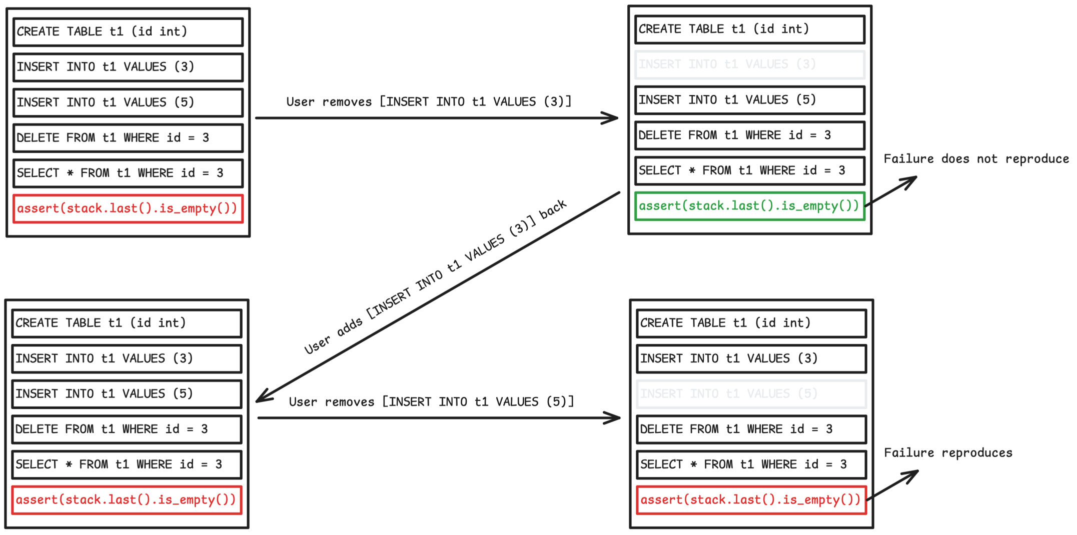
CREATE TABLE t1 (id int)
INSERT INTO t1 VALUES (3)
INSERT INTO t1 VALUES (5)
DELETE FROM t1 WHERE id = 3
SELECT * FROM t1 WHERE id = 3
assert(stack.last().is_empty())
User removes [INSERT INTO t1 VALUES (3)]
Failure does not reproduce
User adds [INSERT INTO t1 VALUES (3)] back
User removes [INSERT INTO t1 VALUES (5)]
Failure reproduces

Figure 4: Interactive Shrinking as User Actions

5.1 RQ1: Does D4 find bugs in SomeDB?

We have collected over 23 confirmed unique bugs found by D4 in SomeDB over the course its development, evolving in complexity as SomeDB grows. At the time of our submission, 19 of these 23 bugs have been fixed. Table 1 is the list of confirmed bugs, along with the changes to the testing infrastructure that resulted in the discovery. The bugs range from simple parser level bugs that could be classified as minor problems, to severe logic bugs deep in the core functionality of the database that could result in data loss. Together with with the list and brief description of bugs presented in Table 1, we choose three representative bugs as case studies, detailing the conditions in which they were discovered and triggered. We selected the three cases to demonstrate the diversity of bugs found in SomeDB with respect to their origin (the bytecode compiler, the B-tree index, and the database header), to how they were discovered and reported (automated report, ourselves, and the developers), and to the oracles used (simple deletion properties, large indexed table generation, and developer-written properties).

Case 1: DELETE not emitting constant Where terms. Our first case study concerns a bug in the compiler of SomeDB. SomeDB translates SQL statements into bytecode and then runs that bytecode in a virtual machine (this is the same approach that SQLite uses [9]). The bytecode compilation is error-prone, as small changes to the compiled SQL expression can greatly affect the generated bytecode.

The bug stemmed from an error in the bytecode compilation of `DELETE`, specifically in the case of constant expressions in the `WHERE` terms. Constant expressions can be compiled to run before the main execution loop of the query, but instead the compilation step just skipped them. This caused `DELETE` with a constant that evaluates to `FALSE` to be incorrectly executed. Below is an instance of the bug manifesting for `DELETE FROM t WHERE 5-5`. The provided bytecode sequence, obtained using SQLite's `explain` directive, shows the correct result. The bug stemmed from the bytecode at addresses 2, 9 and 10 not being emitted when the `WHERE` clause of `DELETE` was composed of constants.

```
> explain DELETE FROM t WHERE 5-5;
addr  opcode        p1  p2  p3  comment
----  ------------  --  --  --  -------
0     Init          0   8   0  Start at 8
1     OpenWrite     0   2   0  root=2; t
2     IfNot         1   7   1  if !r[1] goto 7
3     Rewind        0   7   0  Rewind table t
4       RowId       0   3   0  r[3]=t.rowid
5       Delete      0   0   0
6     Next          0   4   0
7     Halt          0   0   0
8     Transaction   0   1   0  write=true
9     Integer       5   2   0  r[2]=5
10    Subtract      2   2   1  r[1]=r[2]-r[2]
11    Goto          0   1   0
```

This bug was automatically reported by D4 running in SomeDB CI and fixed within the week. The bug was discovered as a result of our the PQS implementation in SomeDB augmented with validity preserving queries between the `INSERT` and `SELECT` statements. The random tester added a naive `DELETE` operation that should not have affected the results of the `SELECT`, keeping the containment property intact, but the assertion that the inserted row should be in the result of the `SELECT` failed, triggering the bug report.

Case 2: Interior node replacement caused self-reference when depth exceeded 2. Our second case study focuses on a bug in the balancing logic of the B-tree implementation of SomeDB. As mentioned throughout the paper, we have paced the generated interactions to match the state of the SomeDB development at the time. While we were only generating very simple `CREATE`, `INSERT`, and `SELECT` statements at the beginning, we added `DELETE` and `UPDATE` as they were added to the database. More recently, SomeDB added database indexes as an experimental feature, at which point we extended generation to support statements such as `SELECT DISTINCT`, `CREATE INDEX`, and compound operators such as `UNION` or `UNION ALL` to the space of generated inputs, and started generating larger tables to take advantage of the indexes. This resulted in triggering a crash failure in the core B-tree data structure that could result in data loss or corruption if not fixed. We reported this bug to the SomeDB developers, and the bug is currently fixed.

Case 3: Database Header Initialization. Our third case study highlights the importance of considering fault scenarios. SQLite uses write-ahead logs (WAL) in order to provide non-blocking reads and ACID transactions in the database. This strategy incurs an additional complexity to the persistent state of the DBMS as the database file might be out-of-date for certain operations, requiring the DBMS to read such information from WAL. This bug is the result of such a situation, where metadata like database size and schema is out-of-date at the time of reading, leading to incorrect operations. D4 found this bug through an assertion failure in the freelist structure in the database header.

As a result, the developer reporting the original bug has written the first example of a *regression property*, added a new fault primitive `ReopenDatabase` that closes down existing connections with the database and reopens them later, enabling the detection of any future bugs that might be a result of an error in the persistent state logic of SomeDB.

5.2 RQ2: How does D4 compare to SQLancer in terms of finding actionable bugs?

In order to answer RQ2, we compare the rate of true positives and false positives reports from three different random testing configurations: D4, the default SQLite3 integration of SQLancer with pragmas disabled (denoted SQLancer-SQLite), and the fork of a work-in-progress SomeDB integration of SQLancer by a SomeDB contributor (denoted SQLancer-SomeDB). Pragmas had to be removed for SQLancer-SQLite because most of them were unimplemented in SomeDB, causing all reported bugs to be false positives in our testing.

We define an actionable bug report from the tools as a sequence of SQL statements that *would have been* submitted and confirmed as a bug at the time of its original report. We used the 23 bug reports we analyzed in the previous section as the ground truth, finding commits which are known to have bugs. The 23 bugs span over 20 SomeDB commits, six of which do not provide the Java bindings necessary for connecting with SQLancer. In the remaining 14 commits in the commit history, we ran all three tools, D4, SQLancer-SQLite, and SQLancer-SomeDB, 100 times.

We analyzed the results of each run and separated them into three bins: *True Positives* that would have been submitted and confirmed as bugs, *False Positives* that are reported as failures by the tools but do not constitute bugs as they are unimplemented features or bugs in the generation, and *No Bugs* that did not report a failure, provided in the stacked bar chart in Fig. 5.

As seen in the figure, SQLancer-SQLite achieves a true positive rate of less than 10% for all 14 commits. SQLancer-SomeDB has much lower false positive rate on average (58%), with a 42% true positive rate that is much closer to D4's 63%. The first difference between D4 and SQLancer-SomeDB lies in the false positives, with D4 under 1% against SQLancer-SomeDB's 58%. The major difference is the number of *actionable bugs*, calculated by de-duplicating the reasons for failure in each true positive. In comparison to the 25 unique bugs found by D4, SQLancer-SQLite and SQLancer-SomeDB found six unique bugs in total not found by the D4, which we have also submitted as separate issues (described below).

These results do not imply that D4 has a better generator or a better set of oracles than SQLancer. Rather, they demonstrate that for actively developed databases with many missing features, using SQLancer is simply not feasible due to its comprehensiveness. D4, by the virtue of developing with the database, finds more bugs.

Bugs Found by SQLancer.

- (`UPDATE t SET (c0, c0)=(0, 0)`): This expression results in a "Column specified more than once" error that has not been fixed at the time of our submission.
- (`SELECT (0x0)`): There was a parsing error for hexadecimals that resulted in an "invalid float literal" error that has since been fixed before our experiments with SQLancer.
- (`SELECT * FROM t WHERE c0 GLOB c0`): There was a logic bug in the `GLOB` that panicked when the values passed were not `TEXT`. Executing this statement after inserting a `NULL` triggered the bug, which was fixed soon after we reported it.
- (`INSERT INTO t VALUES ("a")`): Using double quotes for string literals is discouraged, and SQLite3 even provides a runtime flag for disabling it. We reported this bug as SomeDB panicked for the provided statement.
- (`INSERT INTO t VALUES ((0 BETWEEN 0 AND 0)), (0)`): Constant `BETWEEN` expressions were not rewritten when used within `INSERT`, but the query compiler expected them to be rewritten. We reported this bug as SomeDB panicked for the provided statement.
- (`INSERT INTO t(c2, c0) VALUES (0, 0), (0, 0)`): There was a bug in calculating the column indexes when inserting values to a table in reverse order. SQLancer discovered the bug when the initial table `t` had a `NOT NULL` clause for the column `c0`, which promptly failed after executing the statement. We reported this bug after manually inspecting the result of the failure.

In all six cases, we see the benefit of generating the entire possible input space for SQL dialect of SQLite, which D4 does not, hence missing such bugs. Our evaluation demonstrates the benefit of such comprehensive generation, but also shows that without tailoring to the database, the produced bug reports will be overwhelmingly dominated by false positives and reproductions of existing bugs yet to be fixed.

5.3 RQ3: Does interactive shrinking benefit to counter-example minimization on top of automated shrinking?

While we have not yet run a formal user study to measure the effects (see §7), we have anecdotal evidence that D4's approach to shrinking is effective. Multiple SomeDB developers have been using the interactive shrinker successfully, and the authors have

Bug Id	Description	Oracle	Module	Status
466	TRUE not accepted as catch-all predicate	No Error	Query Compiler	Fixed
548	Infinite loop when checkpointing on Linux	No Infinite Loop	I/O Subsystem	Fixed
629	Query Optimizer broke TRUE in predicates	No Panic	Query Optimizer	Fixed
662	SELECT with nested Boolean expressions sometimes gave no results	PQS	Query Optimizer	Fixed
681	Storage Engine (B-tree) insert caused subtract with overflow	No Panic	Storage Engine (B-tree)	Fixed
682	Faulty recursive binop logic caused SELECT to miss rows	PQS	Query Compiler	Fixed
924	Storage Engine (B-tree) balancing caused page corruption when deleting	No Panic	Storage Engine (B-tree)	Fixed
1040	LIKE operator did not work for non-text values	No Panic	Query Executor	Fixed
1203	Storage Engine (B-tree) balancing error	No Panic	Storage Engine (B-tree)	Fixed
1629	Storage Engine (B-tree) cell updates caused infinite loop in UPDATE	No Infinite Loop	Storage Engine (B-tree)	Fixed
1734	DELETE did not emit conditional jumps if WHERE term was constant	Delete-Select	Query Compiler	Fixed
1815	Storage Engine (B-tree) failed to balance when insert caused cell overflow	No Panic	Storage Engine (B-tree)	Fixed
1818	Always read DB header and schema from file instead of memory page	No Panic	Page Manager	Fixed
1975	Storage Engine (B-tree) expected table or index leaf page	No Panic	Storage Engine (B-tree)	Fixed
1991	Use after free when validating B-tree balance	No Panic	Storage Engine (B-tree)	Fixed
2024	SELECT ... LIMIT resulted in different rows from SQLite	Differential	Query Executor	Open
2026	UPDATE then SELECT resulted in different rows from SQLite	Differential	Query Executor	Open
2047	Overflow cell with divider cell was not found due to faulty validation	No Panic	Storage Engine (B-tree)	Fixed
2074	SELECT hung with long text, CacheFull error	No Panic	Page Manager	Open
2075	Large table with 128 columns handled incorrectly	No Panic	Page Manager	Open
2088	Incorrect record header size calculation	No Panic	Page Manager	Fixed
2106	Interior node replacement caused self-reference when depth exceeded 2	No Panic	Storage Engine (B-tree)	Fixed
2116	Advance after post-delete balancing did not advance B-tree cursor	No Panic	Storage Engine (B-tree)	Fixed

Table 1: List of confirmed unique SomeDB bugs found and reported by D4

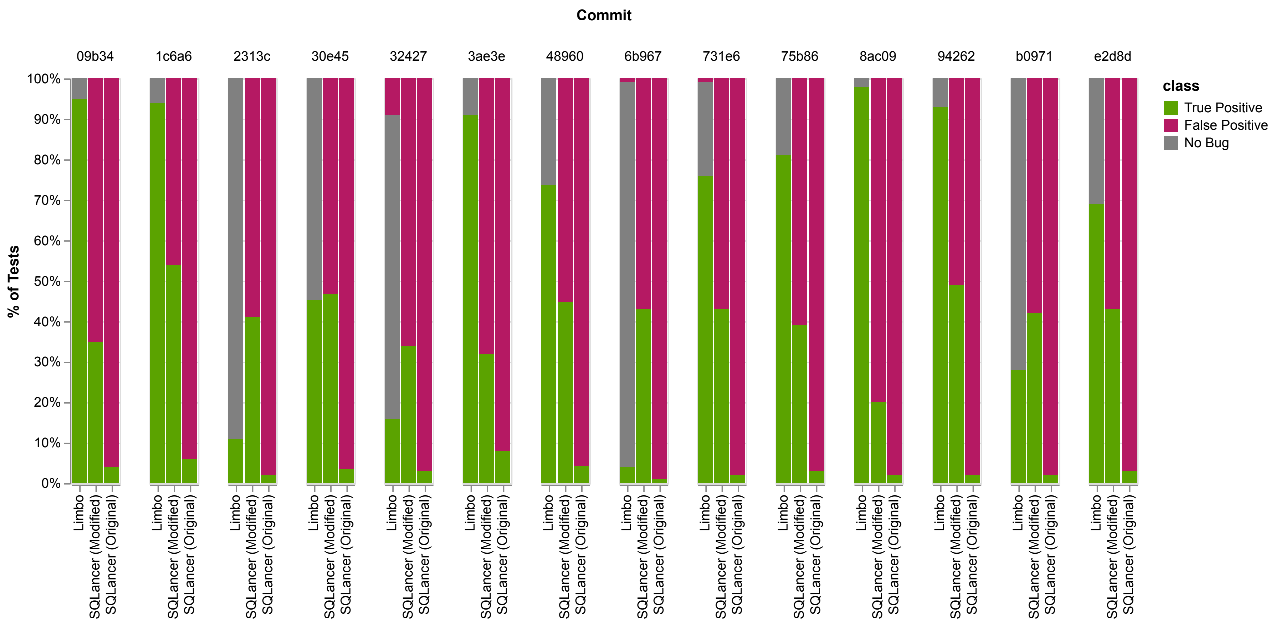

Figure 5: Comparison of True Positives, False Positives and No Bugs for D4, SQLancer-SQLite, SQLancer-SomeDB

also used the interactive shrinker for minimizing counterexamples that were reported to SomeDB.

A brief analysis of the bug reports demonstrates that: for bug ID 924, we were able to interactively shrink the bug to five interactions from its starting point of 1000. For bug ID 1975, automated shrinking reduced the original 1000 interactions to 40 interactions, which developers were able to reduce down to 12 interactions via interactive shrinking. Lastly, in bug ID 2047, the 20 interactions that were produced from automatic shrinking were reduced to 11 interactions.

Our answer to RQ3, therefore, is positive but not yet conclusive.

6 DISCUSSION

D4 is a paradigm for testing databases during development, even though we use D4 the paradigm and D4 for SomeDB interchangeably throughout the paper. We compare the performance of D4 for SomeDB against SQLancer with minimal modifications, which begets some questions worth discussing.

Why not integrate SQLancer to the development process instead of developing a random testing framework from scratch? As we briefly discussed in previous sections, SQLancer is not designed for the constant evolution of the random testing infrastructure along with the project. It is a massive project with its own trajectory, developments, new oracles that can reveal entirely new classes of bugs. D4 aims to empower the database developers themselves by giving them a mechanism to debug the database, instead of giving them bugs to solve. There are additional practical barriers. For example, the Java bindings for SomeDB are not complete at the moment, so we had to fix bugs in the SQLancer-SomeDB integration implemented by one of the SomeDB developers that relied on incomplete features that silently failed. Panics in SomeDB also caused SQLancer to terminate, so it was not possible to shrink the counterexamples with SQLancer reducers without further changes to the integration.

Given that SQLancer supports expected errors for reducing false positives, could we use expected errors for known bugs and false positives we found to discover more bugs? We could use expected errors, and we would have discovered more bugs in the process. In terms of any static evaluation target, tweaking SQLancer by progressively finding more bugs as found ones are marked as expected, it is always possible to make SQLancer find more bugs. The point, however, is that constant modification is not the expected and supported mode of operation when using SQLancer. Although it is phenomenal for off-the-shelf usage, its capabilities unfortunately act as a disadvantage against its use in actively developed projects with many missing features. We have shown that a project with a much smaller scope can adopt the ideas of input generation and oracles in SQLancer as well as other related work on database testing such as Apollo [12] or Thanos [7]. We also have demonstrated that database developers can turn their domain expertise into writing properties as presented in the third case study in §5.1.

Additionally, during our evaluations as well as through the bug reports we submitted, we observed that many crash failures are the result of inline assertions in SomeDB. These crashes directly imply logic bugs as they contradict invariants specified by the developers, which blurs the distinction between crash failures and logic bugs in the literature. Such assertions do not completely replace SQLancer oracles or SomeDB properties because they do not have the ability to follow values through execution in a holistic way as properties do, so they can only reason about local invariants.

Role in SomeDB. As a last point of discussion, we would like to clarify our role in the implementation of D4 in SomeDB, in order to fully credit the open source contributors for their work. At the time we started working on SomeDB, it had a small, unstructured random testing infrastructure that overwhelmingly focused on being able to support Deterministic Simulation Testing [28]. I/O was implemented in a way that could be easily simulated, which allowed for `FAULT`s to be integrated in the generated interactions. We have taken a role as an external open source contributor to the project, gradually proposing improvements to random testing infrastructure such as the Generation Actions DSL, most of the existing properties, stateful random generation, heuristic, brute-force and interactive shrinking, additional oracles such as differential testing against SQLite and determinism checking, and have implemented such proposals. The infrastructure has grown outside of our control at times, producing regressions through the development, along with many improvements by the maintainers and contributors to the project that contributed to the list of bugs found in Table 1. The SQLancer-SomeDB integration was almost entirely developed by one contributor, which we have used in our evaluations with small changes to their code.

7 CONCLUSION AND FUTURE WORK

In this paper, we introduced D4, a new paradigm and concrete implementation for debugging databases during development. By tightly integrating the testing infrastructure with the DBMS itself, D4 avoids many of the limitations that off-the-shelf testing frameworks like SQLancer face when applied to rapidly evolving systems with incomplete feature sets. Our framework provides developers with tools to express custom correctness properties via generation actions, ensures state-aware query generation, and incorporates human-in-the-loop shrinking to make counterexamples actionable.

We implemented D4 for SomeDB, an open-source, actively developed OLTP engine compatible with SQLite3, and demonstrated its effectiveness in practice. D4 discovered 23 unique confirmed bugs, with nearly all of them resulting in actionable reports and minimal false positives. In contrast, state-of-the-art testing frameworks struggled with high false positive rates in the same setting.

Our experience shows that when databases are still under heavy development, flexible, integrated testing approaches that evolve alongside the system are significantly more productive and developer-friendly than external, comprehensive tools. Moreover, our approach fosters collaboration between testing infrastructure and DBMS development by giving developers intuitive ways to encode domain-specific knowledge as properties.

Looking forward, we plan to extend D4 in several directions. First, we are exploring novel methods to stateful shrinking based on generation actions, as well as running a comprehensive user study that will measure the effects of different shrinking strategies on debugging performance. Second, we aim to expand our support for fault injection, enabling simulation of more realistic and chaotic failure conditions. Third, we are interested in incorporating feedback-guided generation strategies, borrowing ideas from coverage-guided fuzzing to improve exploration. Finally, we plan to develop a module for concurrency testing, which remains a significant challenge in DBMS correctness. Additionally, we would like to continue developing the random testing infrastructure as SomeDB evolves, add new generators that explore previously unexplored parts of the input space, as well as adding new oracles taken from the literature on database testing.

D4 bridges the gap between random testing and practical debugging by empowering developers to guide, understand, and act on test results. We believe this paradigm has broad applicability beyond SomeDB and offers a promising path forward for testing other evolving and complex systems.

REFERENCES

[1] Ba, J., and Rigger, M. Testing database engines via query plan guidance. In *Proceedings of the 45th International Conference on Software Engineering* (2023), ICSE '23, IEEE Press, p. 2060–2071.

[2] Ba, J., and Rigger, M. Cert: Finding performance issues in database systems through the lens of cardinality estimation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024), ICSE '24, Association for Computing Machinery.

[3] Ba, J., and Rigger, M. Keep it simple: Testing databases via differential query plans. *Proc. ACM Manag. Data 2*, 3 (May 2024).

[4] Böhme, M., Pham, V.-T., and Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, Association for Computing Machinery, p. 1032–1043.

[5] Claessen, K., and Hughes, J. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not. 35*, 9 (Sept. 2000), 268–279.

[6] Fioraldi, A., Maier, D., Eissfeldt, H., and Heuse, M. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)* (Aug. 2020), USENIX Association.

[7] Fu, Y., Wu, Z., Zhang, Y., Liang, J., Fu, J., Jiang, Y., Li, S., and Liao, X. Thanos: DBMS Bug Detection via Storage Engine Rotation Based Differential Testing . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)* (Los Alamitos, CA, USA, May 2025), IEEE Computer Society, pp. 655–666.

[8] Goldstein, H., Cutler, J. W., Dickstein, D., Pierce, B. C., and Head, A. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024), ICSE '24, Association for Computing Machinery.

[9] Hipp, R. D. SQLite, 2020.

[10] Hritcu, C., Hughes, J., Pierce, B. C., Spector-Zabusky, A., Vytiniotis, D., Azevedo de Amorim, A., and Lampropoulos, L. Testing noninterference, quickly. *SIGPLAN Not. 48*, 9 (Sept. 2013), 455–468.

[11] Hughes, J. How to specify it! a guide to writing properties of pure functions. In *Trends in Functional Programming: 20th International Symposium, TFP 2019, Vancouver, BC, Canada, June 12–14, 2019, Revised Selected Papers* (Berlin, Heidelberg, 2019), Springer-Verlag, p. 58–83.

[12] Jung, J., Hu, H., Arulraj, J., Kim, T., and Kang, W. APOLLO: automatic detection and diagnosis of performance regressions in database systems. *PVLDB 13*, 1 (2019), 57–70.

[13] Lampropoulos, L., Hicks, M., and Pierce, B. C. Coverage guided, property based testing. *Proc. ACM Program. Lang. 3*, OOPSLA (Oct. 2019).

[14] Lampropoulos, L., Paraskevopoulou, Z., and Pierce, B. C. Generating good generators for inductive relations. *Proc. ACM Program. Lang. 2*, POPL (Dec. 2017).

[15] Lemieux, C., Padhye, R., Sen, K., and Song, D. Perffuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2018), ISSTA 2018, Association for Computing Machinery, p. 254–265.

[16] Maciver, D., and Donaldson, A. F. Test-case reduction via test-case generation: Insights from the hypothesis reducer (tool insights paper). In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)* (2020), R. Hirschfeld and T. Pape, Eds., vol. 166 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 13:1–13:27.

[17] Padhye, R., Lemieux, C., Sen, K., Papadakis, M., and Le Traon, Y. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2019), ISSTA 2019, Association for Computing Machinery, p. 329–340.

[18] Padhye, R., Lemieux, C., Sen, K., Simon, L., and Vijayakumar, H. Fuzzfactory: domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang. 3*, OOPSLA (Oct. 2019).

[19] Rigger, M., and Su, Z. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2020), ESEC/FSE 2020, Association for Computing Machinery, p. 1140–1152.

[20] Rigger, M., and Su, Z. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang. 4*, OOPSLA (Nov. 2020).

[21] Rigger, M., and Su, Z. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 667–682.

[22] Rigger, M., and Su, Z. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation* (USA, 2020), OSDI'20, USENIX Association.

[23] Seltenreich, A. Sqlsmith. https://github.com/anse1/sqlsmith, 2015.

[24] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, Association for Computing Machinery, p. 283–294.

[25] Zeller, A. *The Debugging Book.* CISPA Helmholtz Center for Information Security, 2024. Retrieved 2024-07-01 16:49:37+02:00.

[26] Zeller, A., and Hildebrandt, R. Simplifying and isolating failure-inducing input. 183–200.

[27] Zhang, C., and Rigger, M. Constant optimization driven database system testing. *Proc. ACM Manag. Data 3*, 1 (Feb. 2025).

[28] Zhou, J., Xu, M., Shraer, A., Namasivayam, B., Miller, A., Tschannen, E., Atherton, S., Beamon, A. J., Sears, R., Leach, J., Rosenthal, D., Dong, X., Wilson, W., Collins, B., Scherer, D., Grieser, A., Liu, Y., Moore, A., Muppana, B., Su, X., and Yadav, V. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data* (New York, NY, USA, 2021), SIGMOD '21, Association for Computing Machinery, p. 2653–2666.