DIRT: Database-Integrated Random Testing

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 Buffalo
Buffalo, USA
hgoldste@buffalo.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 DIRT, a paradigm designed specifically for testing 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. We evaluate DIRT on Turso, 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.

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, with approaches ranging from binary fuzzing with AFL [6] to structured query generation in SQLSmith [24], and to the powerful test oracles of SQLancer [22]. 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 testing 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 study this problem in the context of Turso, an open-source SQLite3-compatible OLTP database engine in active development. With an average of more than 30 commits a day, Turso is a rapidly evolving target for testing. DIRT helped find 23 confirmed bugs, all of which were subsequently fixed. In comparison, running SQLancer-SQLite3 with minimal modifications in Turso led to a false positive rate of 96.5% with only one new bug, and running SQLancer-Turso, an existing independently developed SQLancer integration for Turso, without modifications led to a false positive rate of 58% with six newly discovered bugs.

In summary, our contributions are as follows:

(1) We propose DIRT, a paradigm for testing databases during development which integrates a testing framework within the database and provides convenient and flexible abstractions to developers.
(2) We implement DIRT for Turso, an open source SQLite3-compatible OLTP database under active development.
(3) We evaluate the effectiveness of DIRT by finding actionable bugs in Turso: we found 23 unique confirmed bugs with very few false positives compared to the state-of-the-art.

2 BACKGROUND AND RELATED WORK

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

2.1 Fuzzing and Property-Based Testing

In Fuzz Testing [25], the primary focus is on smart and effective test-case generation that can work out of the box with minimal user inputs. The key idea is to leverage *runtime feedback*, usually in the form of branch coverage [4, 6], though not always [19], 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 simple: does the program crash? Other oracles include fuzzing against a model or a different implementation (known as differential fuzzing [9, 13]) or checking for excessive execution times [16].

Property-Based Testing (PBT), on the other hand, primarily focuses on empowering users to write their own oracles [5, 12]. 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, 15]. 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 [14, 18], and in the context of database testing, both SQLancer and DIRT follow such a hybrid viewpoint.

2.2 SQLancer

SQLancer is a multi-year research project by Rigger et. al that 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) [23], 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) [20], which found 51 optimization bugs, and Ternary Logic Partitioning (TLP) [21], which discovered 77 novel logic bugs.

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 join optimizations, and Constant Optimization Driven Database System Testing (CODDTest) [26] for finding logic bugs in optimizations. 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 process. 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.

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. DIRT 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 Turso for reducing false positives, but also tailoring the correctness oracles to new features as they are added, decreasing false negatives.

In this section, we introduce *generation actions*, an imperative formulation for properties that do not just describe *what* the property tests, but *how* to test it. We begin by using a very simple commutativity property to convey the basic ideas and notation, then detail the oracles that were actually developed to test Turso, and conclude the section with a description of our generation strategy. For concreteness, let us consider the following equivalence relation leveraging the commutativity of ($\land$): for any database db and any two predicates p and q that can range over variables from the database, if we evaluate $SELECT\ (p \land q)$ and $SELECT(q \land p)$ in db they should yield equivalent results:

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

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). On the right, we express the same high-level commutativity property as a generation action that returns a `property` parameterized by a context that contains a database `db`. Rather than independently generating p and q, we fix some of the details of generation. We `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`. For `q`, we generate an arbitrary `expression` that can refer to both `t` and `c`. We use ⟵ to bind the results of generation and := for regular let-style binding. We 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.

```
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)
```

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

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

```
gen property db:
    (t1, t2) ⟵ (pick db.tables, pick db.tables)
    (c1, c2) ⟵ (pick t1.columns, pick t2.columns)
    (r1, r2) ⟵ (gen row t1, 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).

DIRT as part of the DSL for generation actions, as exemplified by the header initialization bug studied in §4.1.

3.1 Definitions of Oracles in DIRT

In this subsection, we go over definitions of five 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) [23], 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, modeling the implementation of PQS that tests Turso 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 DIRT. Fig. 2a demonstrates how to write Non-Optimizing Reference Engine Construction Generation (NoREC) [20] as a GA. Fig. 2b presents the GA for `WHERE Extended` case of Ternary Logic Partitioning (TLP) [21] 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.

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 [11], 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 [12], 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 DIRT 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, describing the proportion of read, write, and create instructions that should appear in the generated interactions. The algorithm iteratively generates one or more interactions compatible with the

```
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"

Figure 2: Definitions of Oracles as Generation Actions

current shadow state, updates the state to account for the new interactions, and repeats until enough interactions have been generated.

SQLancer does not keep a separate state [23], 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 EVALUATION

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

- RQ1: Does DIRT find bugs in Turso?
- RQ2: How does DIRT compare to the state-of-the-art?

4.1 RQ1: Does DIRT find bugs in Turso?

We have collected over 23 confirmed and fixed unique bugs found by DIRT in Turso over the course its development, evolving in complexity as Turso grows. 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. We choose three representative bugs as case studies, detailing the conditions in which they were discovered and triggered. The cases demonstrate the diversity of bugs found in Turso 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. The first bug lies in the compiler of Turso. Turso translates SQL statements into bytecode and then runs that bytecode in a virtual machine (this is the same approach that SQLite uses [10]). 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.

This bug was automatically reported by DIRT running in Turso CI and fixed within the week. The bug was discovered as a result of our the PQS implementation in Turso 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. The second bug manifested in the balancing logic of the B-tree implementation of Turso, but only because the generated interactions were paced to match the state of the Turso development. When Turso added database indexes as an experimental feature, 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

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

Table 1: List of confirmed unique Turso bugs found and reported by DIRT

loss or corruption if not fixed. We reported this bug to the Turso developers, and the bug is currently fixed.

Case 3: Database Header Initialization. The third selected bug 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. DIRT 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 Turso.

4.2 RQ2: How does DIRT compare to SQLancer?

We answer RQ2 by comparing the rate of true positives and false positives reports from three different random testing configurations: DIRT, the default SQLite3 integration of SQLancer with pragmas disabled (denoted SQLancer-SQLite), and the fork of a work-in-progress Turso integration of SQLancer by a Turso contributor (denoted SQLancer-Turso). Pragmas had to be removed for SQLancer-SQLite because most of them were unimplemented in Turso, 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 as the ground truth, finding commits which are known to have bugs. The bugs span over 20 Turso commits, 6 of which do not provide Java bindings necessary for connecting with SQLancer. We ran all three tools on the remaining 14 commits in the commit history. 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, and *No Bugs* that did not report a failure, provided in the stacked bar chart in Fig. 3. SQLancer-SQLite achieves a true positive rate below 10% across all 14 commits. SQLancer-Turso performs better, with a 42% true positive rate, but still yields many more false positives than DIRT (58% vs. under 1%). DIRT also finds substantially more actionable bugs: 25 unique bugs, compared with six found by the two SQLancer configurations combined (§A.1). This does not mean that DIRT has better generators or oracles than SQLancer. Rather, it shows that for rapidly evolving databases with many missing features, database-integrated testing yields more actionable bugs.

5 DISCUSSION

DIRT is a paradigm for testing databases during development, even though we use DIRT the paradigm and DIRT for Turso interchangeably throughout the paper. We compare the performance of DIRT for Turso 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 earlier, SQLancer is not designed for the constant evolution of the random testing infrastructure along with the project. It is a project with its own trajectory, development, new algorithms,

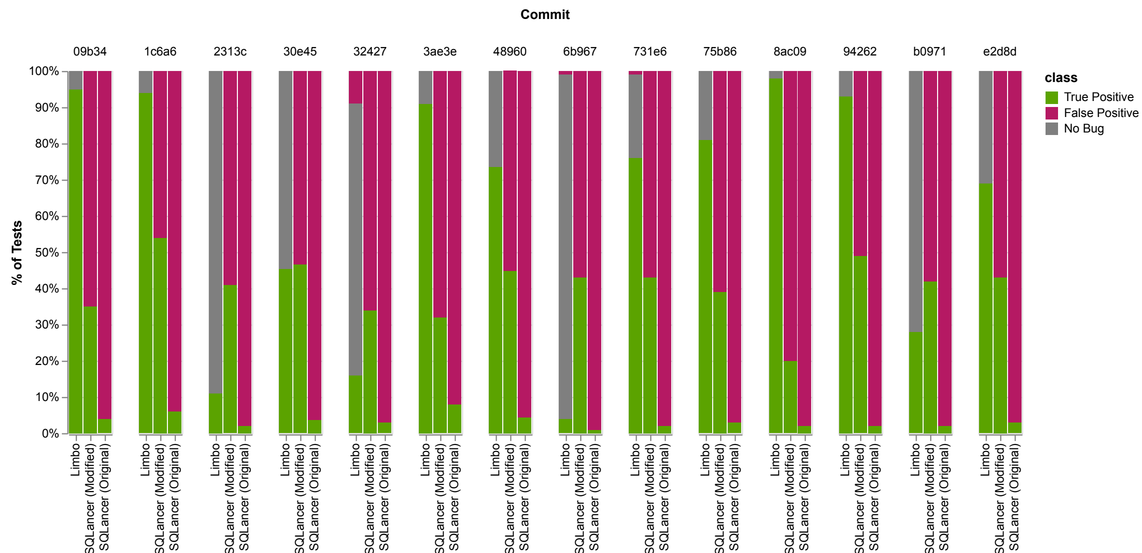

Figure 3: Comparison of True Positives, False Positives and No Bugs for DIRT, SQLancer-SQLite, SQLancer-Turso

generators, and oracles. DIRT aims to empower the database developers themselves by giving them a mechanism to test the database, instead of giving them bugs to solve. There are additional practical barriers; Java bindings for Turso are not complete at the moment, so we had to fix bugs in the SQLancer-Turso integration implemented by one of the Turso developers that relied on incomplete features that silently failed, panics in Turso caused SQLancer to terminate, so it was not possible to minimize the counterexamples with SQLancer reducers without further changes.

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 [13] or Thanos [8] and demonstrated that database developers can turn their domain expertise into writing properties as presented in the third case study in §4.1. We also observed that many crash failures are the result of inline assertions in Turso. These crashes imply logic bugs as they contradict invariants specified by the developers, blurring the distinction between crash failures and logic bugs in the literature. Such assertions do not replace SQLancer oracles or Turso 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 Turso. As a last point of discussion, we would like to clarify our role in the implementation of DIRT in Turso, in order to fully credit the open source contributors for their work. At the time we started working on Turso, 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, 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-Turso integration was almost entirely developed by one contributor, which we have used in our evaluations with small changes to their code.

Related work on adaptive DBMS testing. Although we compared DIRT mainly against SQLancer as the basis of our oracles, other systems such as SQLRight [17], Griffin [7], and SQLancer++ [27] also adapt testing to DBMS behavior. However, these systems primarily approach adaptation from the perspective of an external testing framework, with evaluations centered on broad effectiveness across multiple databases and minimal manual intervention. In contrast, DIRT studies adaptation as a database-integrated process: the testing infrastructure evolves together with a single DBMS under development, and developers directly shape generators, properties, and fault models as the implementation changes.

6 CONCLUSION AND FUTURE WORK

In this paper, we introduced DIRT, a new paradigm and concrete implementation for testing databases during development. By tightly integrating the testing infrastructure with the DBMS itself, DIRT 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 and ensures state-aware query generation. We implemented DIRT for Turso and found 23 confirmed bugs with minimal false positives. In the same setting, SQLancer-based baselines produced substantially more false positives and fewer actionable reports. 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.

DIRT 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 Turso and offers a promising path forward for testing other evolving and complex systems.

A APPENDIX

A.1 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 Turso 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 Turso 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 DIRT 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.

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, J., Liang, J., Wu, Z., Wang, M., and Jiang, Y. Griffin : Grammar-free dbms fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2023), ASE '22, Association for Computing Machinery.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] Liang, Y., Liu, S., and Hu, H. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *Proceedings of the 31st USENIX Security Symposium (USENIX 2022)* (Boston, MA, aug 2022).

[18] 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.

[19] 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).

[20] 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.

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

[22] 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.

[23] 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.

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

[25] Zeller, A., Gopinath, R., Böhme, M., Fraser, G., and Holler, C. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. Retrieved 2024-07-01 16:50:18+02:00.

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

[27] Zhong, S., and Rigger, M. Scaling automated database system testing, 2026.

[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.