Testing Database Engines via Pivoted Query Synthesis

Manuel Rigger and Zhendong Su, *ETH Zurich*

https://www.usenix.org/conference/osdi20/presentation/rigger

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Testing Database Engines via Pivoted Query Synthesis

Manuel Rigger Zhendong Su
Department of Computer Science, ETH Zurich

Abstract

Database Management Systems (DBMSs) are used widely, and have been extensively tested by fuzzers, which are successful in finding crash bugs. However, approaches to finding *logic bugs*, such as when a DBMS computes an incorrect result set, have remained mostly untackled. To this end, we devised a novel and general approach that we have termed *Pivoted Query Synthesis*. The core idea of this approach is to automatically generate queries for which we ensure that they fetch a specific, randomly selected row, called the *pivot row*. If the DBMS fails to fetch the pivot row, the likely cause is a bug in the DBMS. We tested our approach on three widely-used and mature DBMSs, namely SQLite, MySQL, and PostgreSQL. In total, we found 121 unique bugs in these DBMSs, 96 of which have been fixed or verified, demonstrating that the approach is highly effective and general. We expect that the wide applicability and simplicity of our approach will enable improving the robustness of many DBMSs.

1 Introduction

Database management systems (DBMSs) based on the relational model [10] are a central component in many applications, since they allow efficiently storing and retrieving data. They have been extensively tested by random query generators such as SQLsmith [45], which have been effective in finding queries that cause the DBMS process to crash (*e.g.*, by causing a buffer overflow). Also fuzzers such as AFL [2] are routinely applied to DBMSs. However, these approaches cannot detect *logic bugs*, which we define as bugs that cause a query to return an incorrect result, for example, by erroneously omitting a row, without crashing the DBMS.

Logic bugs in DBMSs are difficult to detect automatically. A key challenge for automatic testing is to come up with an effective *test oracle*, that can detect whether a system behaves correctly for a given input [21]. In 1998, Slutz proposed to use *differential testing* [33] to detect logic bugs in DBMSs, by constructing a test oracle that compares the results of a query on multiple DBMSs, which the author implemented in a tool RAGS [46]. While RAGS detected many bugs, differential testing comes with the significant limitation that the systems under test need to implement the same semantics for a given input. All DBMSs support a common and standardized language *Structured Query Language (SQL)* to create, access, and modify data [8]. In practice, however, each DBMS provides a plethora of extensions to this standard and deviates from it in other parts (*e.g.*, in how `NULL` values are handled [46]). This vastly limits differential testing, and also the author stated that the small common core and the differences between different DBMSs were a challenge [46]. Furthermore, even when all DBMSs fetch the same rows, it cannot be ensured that they work correctly, because they might be affected by the same underlying bug.

To efficiently detect logic bugs in DBMSs, we propose a general and principled approach that we termed *Pivoted Query Synthesis (PQS)*, which we implemented in a tool called SQLancer. The core idea is to solve the oracle problem for a single, randomly-selected row, called the *pivot row*, by synthesizing a query whose result set must contain the pivot row. We synthesize the query by randomly generating expressions for `WHERE` and `JOIN` clauses, evaluating the expressions based on the pivot row, and modifying each expression to yield `TRUE`. If the query, when processed by the DBMS, fails to fetch the pivot row, a bug in the DBMS has been detected. We refer to this oracle as the *containment* oracle.

Listing 1 illustrates our approach on a test case that triggered a bug that we found using the containment oracle in the widely-used DBMS SQLite. The `CREATE TABLE` statement creates a new table `t0` with a column `c0`. Subsequently, an index is created and three rows with the values `0`, `1`, and `NULL` are inserted. We select the pivot row `c0=NULL` and construct the random `WHERE` clause `c0 IS NOT 1`. Since `NULL IS NOT 1` evaluates to `TRUE`, we can directly pass the query to the DBMS, expecting the row with value `NULL` to be contained in the result. However, due to a logic bug in the DBMS, the partial index was used based on the incorrect assumption that `c0 IS NOT 1` implied `c0 NOT NULL`, resulting in the pivot row

Listing 1: Illustrative example, based on a *critical* SQLite bug. The check symbol denotes the expected, correct result, while the bug symbol denotes the actual, incorrect one.

```
CREATE TABLE t0(c0);
CREATE INDEX i0 ON t0(1) WHERE c0 NOT NULL;
INSERT INTO t0(c0) VALUES (0), (1), (NULL);
SELECT c0 FROM t0 WHERE c0 IS NOT 1; -- {0}🐞 {0,
    NULL}✔
```

not being fetched. We reported this bug to the SQLite developers, who stated that it existed since 2013, classified it as critical and fixed it quickly. Even for this simple query, differential testing would have been ineffective in detecting the bug. The **CREATE TABLE** statement is specific to SQLite, since, unlike other popular DBMSs, such as PostgreSQL and MySQL, SQLite does not require the column `c0` to be assigned a column type. Furthermore, both MySQL's and PostgreSQL's `IS` **NOT** cannot be applied to integers; they only provide an operator `IS` **DISTINCT FROM**, which provides equivalent functionality. All DBMSs provide an operator `IS` **NOT TRUE**, which, however, has different semantics; for SQLite, it would fetch only the value `0`, and not expose the bug.

To demonstrate the generality of our approach, we implemented it for three popular and widely-used DBMSs, namely SQLite [49], MySQL [36], and PostgreSQL [40]. In total, we found 96 unique bugs, namely 64 bugs in SQLite, 24 bugs in MySQL, and 8 in PostgreSQL, demonstrating that the approach is highly effective and general. 61 of these were logic bugs found by the containment oracle. In addition, we found 32 bugs by causing DBMS-internal errors, such as database corruptions, and for 3 bugs we caused DBMS crashes (*i.e.*, `SEGFAULT`s). One of the crashes that we reported for MySQL was classified as a security vulnerability (CVE-2019-2879). 78 of the bugs were fixed by the developers, indicating that they considered our bug reports useful.

Since our method is general and applicable to all DBMSs, we expect that it will be widely adopted to detect logic bugs that have so far been overlooked. In fact, after releasing a preprint of the paper, we received a number of requests by companies as well as individual developers indicating their interest in implementing PQS to test the DBMSs that they were developing. Among these, PingCAP publicly released a PQS implementation that they have been successfully using to find bugs in TiDB. For reproducibility and to facilitate further research on this topic, we have released SQLancer at https://github.com/sqlancer/. In addition, the artifact associated with the paper contains SQLancer as well as a database of all reported bugs [44]. PQS inspired complementary follow-up work, such as NoREC and TLP, which focus on finding sub-categories of logic bugs [42, 43]. Despite this, PQS has notable limitations; it only partly validates a query's result, and cannot be used, for example, to test aggregate functions, the size of the result set, or its ordering. Furthermore, the effort required to implement the technique depends on the complexity of the operations to be tested, which can be high for complex operators or functions.

In summary, we contribute the following:

- A general and highly-effective approach to finding bugs in DBMSs termed *Pivoted Query Synthesis (PQS)*.
- An implementation of PQS in a tool named SQLancer, used to test SQLite, MySQL, and PostgreSQL.
- An evaluation of PQS, which uncovered 96 bugs.

2 Background

This section provides important background information on relational DBMSs, SQL, and the DBMSs we tested.

Database management systems. We primarily aim to test *relational* DBMSs, that is, those that are based on the *relational data model* proposed by Codd [10]. Most widely-used DBMSs, such as Oracle, Microsoft SQL, PostgreSQL, MySQL, and SQLite are based on it. A relation R in this model is a mathematical relation $R \subseteq S_1 \times S_2 \times ... \times S_n$ where S_1, S_2, ..., S_n are referred to as domains. More commonly, a relation is referred to as a *table* and a domain is referred to as a *data type*. Each tuple in this relation is referred to as a row. SQL [8], a domain-specific language that is based on relational algebra [11], is the most commonly used language to interact with the DBMSs. ANSI first standardized SQL in 1987, and it has since been developed further. In practice, however, DBMSs lack functionality described by the SQL standard and deviate from it. In this paper, we assume basic familiarity with SQL.

Test oracles. An effective *test oracle* is crucial for automatic testing approaches [21]. A test oracle assesses whether a given test case has passed. Manually written test cases encode the programmer's knowledge who thus acts as a test oracle. In this work, we are interested only in automatic test oracles, which would allow comprehensively testing a DBMS. The most successful automatic test oracle for DBMSs is based on *differential testing* [46]. Differential testing refers to a technique where a single input is passed to multiple systems that implement the same language to detect mismatching outputs, which would indicate a bug. In the context of DBMSs, the input corresponds to a database as well as a query, and the systems to multiple DBMSs—when their fetched result sets mismatch, a bug in one of the DBMS would be detected. However, SQL dialects vary significantly, making it difficult to use differential testing effectively. This is also acknowledged by industry. For example, Cockroach Labs state that they *"are unable to use Postgres as an oracle because CockroachDB has slightly different semantics and SQL support, and generating queries that execute identically on both is tricky [...]"* [22]. Furthermore, differential testing is not a *precise* oracle, as it fails to detect bugs that affect all the systems.

Table 1: The DBMSs we tested are popular, complex, and have been developed for a long time.

DBMS	Popularity Rank		LOC	Released
	DB-Engines	Stack Overflow		
SQLite	11	4	0.3M	2000
MySQL	2	1	3.8M	1995
PostgreSQL	4	2	1.4M	1996

Tested DBMSs. We focused on three popular and widely-used open-source DBMSs: SQLite, MySQL, and PostgreSQL (see Table 1). According to the DB-Engines Ranking [1] and the Stack Overflow's annual Developer Survey [38], these DBMSs are among the most popular and widely-used ones. Furthermore, the SQLite website speculates that SQLite is likely used more than all other databases combined; most mobile phones extensively use SQLite, it is used in most popular web browsers, and many embedded systems (such as television sets) [48]. All DBMSs are production-level systems, and have been maintained and developed for about 20 years.

3 Pivoted Query Synthesis

We propose *Pivoted Query Synthesis* as an automatic testing technique for detecting logic bugs in DBMSs. Our core insight is that by considering only a single row at a time, a conceptually-simple test oracle can be created that can effectively detect logic bugs. Specifically, our idea is to select a random row, to which we refer as the pivot row, from a set of tables and views in the database. Subsequently, we randomly generate a set of boolean predicates, which we then modify so that they evaluate to `TRUE` for the values of the pivot row based on an Abstract Syntax Tree (AST) interpreter. By using these expressions in `WHERE` and `JOIN` clauses of an otherwise randomly-generated query, we can ensure that the pivot row must be contained in the result set. If it is not contained, a bug has been found. Basing the approach on an AST interpreter provides us with an exact oracle. While implementing this interpreter requires moderate implementation effort for complex operators (such as regular expression operators), other challenges that a DBMS has to tackle, such as query planning, concurrent access, integrity, and persistence can be disregarded by it. Furthermore, the AST interpreter can be naively implemented without affecting the tool's performance, since it only operates on a single record, whereas the DBMS has to potentially scan through all the rows of a database to process a query.

3.1 Approach Overview

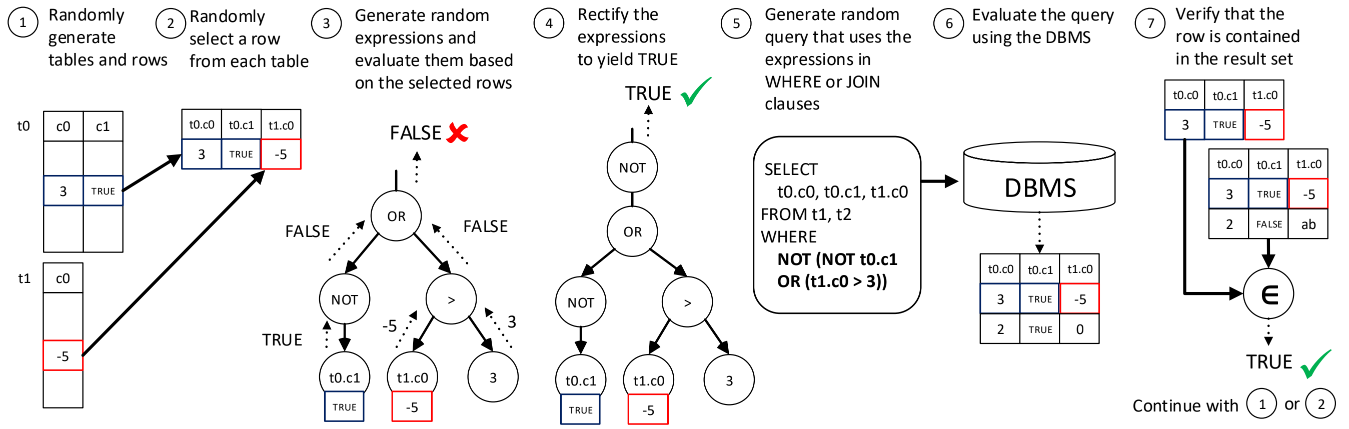
Figure 1 illustrates the detailed steps of PQS. First, we create a database with one or multiple random tables, which we fill with random data (see step ①). We ensure that each table, and randomly generated view, holds at least one row, to enable selecting a random pivot row in step ②. A pivot row is only conceptually a row, and can be composed of columns that refer to rows of multiple tables and/or views. Its purpose is to use it to derive a test case as well as a test oracle to validate the correctness of the DBMS. The pivot row shown in Figure 1 consists of both columns from table `t0` and `t1`. In the next steps, we proceed by constructing a test oracle based on the pivot row. To this end, we randomly create expressions based on the DBMS' SQL grammar and valid table column names (see step ③). We evaluate these expressions, substituting column references by the corresponding values of the pivot row. Then, we modify the expressions so that they yield `TRUE` (see step ④). We use these expressions in `WHERE` and/or `JOIN` clauses for a query that we construct (see step ⑤). We pass this query to the DBMS, which returns a result set (see step ⑥), which we expect to contain the pivot row, potentially among other rows. In a final step, we check whether the pivot row is indeed contained in the result set (see step ⑦). If it is not contained, we have likely detected a bug in the DBMS. For the next iteration, we either continue with step ② and generate new queries for a newly-selected pivot row, or continue with ① to generate a new database.

Our core idea is given by how we construct the test oracle (see steps ② to ⑦). Thus, Section 3.2 first explains how we generate queries and check for containment, assuming that the database has already been created. Section 3.3 then explains step ①, namely how we generate the tables and data. Section 3.4 provides important implementation details.

3.2 Query Generation & Checking

The core idea of our approach is to construct a query for which we anticipate that the pivot row is contained in the result set. We randomly generate expressions to be used in `WHERE` and/or `JOIN` clauses of the query, and ensure that each expression evaluates to `TRUE` for the pivot row. This subsection describes how we generate random predicates that we rectify and then use in the query (*i.e.*, steps ③ to ⑤).

Random predicate generation. In step ③, we randomly generate Abstract Syntax Trees (ASTs) up to a specified maximum depth by constructing a random expression tree based on the database's schema (*i.e.*, the column names and types). For SQLite and MySQL, SQLancer generates expressions of any type, because they provide implicit conversions to boolean. For PostgreSQL, which performs few implicit conversions, the generated root node must produce a boolean value, which we achieve by selecting one of the appropriate operators (*e.g.*, a comparison operator). Algorithm 1 illustrates how generat-



1 Randomly generate tables and rows
2 Randomly select a row from each table
3 Generate random expressions and evaluate them based on the selected rows
4 Rectify the expressions to yield TRUE
5 Generate random query that uses the expressions in WHERE or JOIN clauses
6 Evaluate the query using the DBMS
7 Verify that the row is contained in the result set
t0
c0
c1
3
TRUE
t1
c0
-5
t0.c0
t0.c1
t1.c0
3
TRUE
-5
FALSE ✗
OR
FALSE
FALSE
NOT
>
TRUE
-5
3
t0.c1
TRUE
t1.c0
-5
3
TRUE ✓
NOT
OR
NOT
>
t0.c1
TRUE
t1.c0
-5
3
SELECT
t0.c0, t0.c1, t1.c0
FROM t1, t2
WHERE
NOT (NOT t0.c1
OR (t1.c0 > 3))
DBMS
t0.c0
t0.c1
t1.c0
3
TRUE
-5
2
TRUE
0
t0.c0
t0.c1
t1.c0
3
TRUE
-5
t0.c0
t0.c1
t1.c0
3
TRUE
-5
2
FALSE
ab
∈
TRUE ✓
Continue with 1 or 2

Figure 1: Overview of the approach implemented in SQLancer. Dotted lines indicate that a result is generated.

```
Function generateExpression(int depth):
    node_types ← {LITERAL, COLUMN}
    if depth < maxdepth then
        node_types ← node_types ∪ {UNARY, ...}
    type ← random(node_types)
    switch type do
        case LITERAL do
            return Literal(randomLiteral());
        case COLUMN do
            return Column-
              Value(randomTable().randomColumn());
        case UNARY do
            return
              UnaryNode(generateExpression(depth+1),
              UnaryNode.getRandomOperation());
        case ... do
            ...
    end
```

Algorithm 1: The `generateExpression()` function generates a random AST.

ing the expressions is implemented for MySQL and SQLite. The input parameter `depth` ensures that when a specified maximum depth is reached, a leaf node is generated. The leaf node can either be a randomly-generated constant, or a reference to a column in a table or view. If the maximum depth is not yet reached, also other operators are considered (*e.g.*, a unary operator such as **NOT**). Generating these expressions is dependent on which operators the respective DBMS supports. The random expression generation by itself is not a contribution of this paper; random query generators, such as RAGS [46] and SQLsmith operate similarly [45]. We implemented the expression generators manually for each DBMS under test, based on the respective DBMS SQL dialect's documentation; as part of future work, we will consider automatically deriving them based on the SQL dialect's grammar.

Expression evaluation. After building a random expression tree, we must check whether the condition yields **TRUE** for the pivot row. To this end, every node must provide an `execute()` method that computes the node's result, which needs to be manually implemented. Leaf nodes directly return their assigned constant value. Column nodes are assigned the value that corresponds to their column in the pivot row. For example, in Figure 1 step ③, the leaf node `t0.c1` returns **TRUE**, and the constant node `3` returns an integer `3`. Composite nodes compute their result based on the literals returned by their children. For example, the **NOT** node returns **FALSE**, because its child evaluates to **TRUE** (see Algorithm 2). The node first executes its subexpression, and then casts the result to a boolean; if the result is a boolean value, the value is negated; otherwise **NULL** is returned. Note that our implementation is simpler than AST interpreters for programming languages [50], since all nodes operate on literal values (*i.e.*, they do not need to consider mutable storage). It is also simpler than query engine models, such as the well-known Volcano-style iteration model [16], and widely-used models based on it, such as the vectorized model or the data-centric code generation model, which all need to consider multiple rows [26]. Since the bottleneck of our approach is the DBMS evaluating the queries rather than SQLancer, all operations are implemented naively and do not perform any optimizations. Some operations require moderate implementation effort nevertheless; for example, the implementation of the **LIKE** regular expression operator has over 50 LOC in SQLancer.

Expression rectification. After generating random expressions, step ④ ensures that they evaluate to **TRUE**. SQL is based on a three-valued logic. Thus, when evaluated in a boolean context, an expression either yields **TRUE**, **FALSE**, or **NULL**. To rectify an expression to yield **TRUE**, we use Algorithm 3. For example, in Figure 1 step ④, we modify the expression by adding a preceding **NOT**, so that the expression evaluates to **TRUE**. Note that our approach works also for other logic systems (*e.g.*, four-valued logic), by adjusting this step. Alternatively, it could be checked that the pivot row is ex-

```
Method NotNode::execute():
    $value \leftarrow child.execute()$
    switch asBoolean(value) do
        case TRUE do
            $result \leftarrow FALSE$
        case FALSE do
            $result \leftarrow TRUE$
        case NULL do
            $result \leftarrow NULL$
    end
    return result;
```

Algorithm 2: The execute() implementation of a **NOT** node.

```
Function rectifyCondition(randexpr):
    switch randexpr.execute() do
        case TRUE do
            $result \leftarrow randexpr$
        case FALSE do
            $result \leftarrow NOT\ randexpr$
        case NULL do
            $result \leftarrow randexpr\ ISNULL$
    end
    return result;
```

Algorithm 3: The expression rectification step applied to a randomly-generated expression.

pectedly not contained in the result set, by ensuring that the expression evaluates to **FALSE**.

Query generation. In step ⑤, we generate targeted queries that fetch the pivot row. Most importantly, the expressions evaluating to **TRUE** are used in **WHERE** clauses, which restrict which rows a query fetches, and in **JOIN** clauses, which are used to join tables. Since the expressions evaluate to **TRUE**, the pivot row is guaranteed to be contained in the result set. **JOIN** clauses are not treated specially; as we create the clause's predicate to yield **TRUE** for the pivot row, inner, full, left, and right joins all behave in the same way as a **WHERE** clause with respect to the pivot row. **SELECT** statements typically provide various keywords to control the query's behavior, from which we randomly select applicable options. Specifically, we considered the following elements:

- **DISTINCT** clauses, which filter out duplicate rows, while retaining the guarantee that the pivot row is contained in the result set;
- **GROUP BY** clauses that contain all pivot row columns to guarantee that the pivot row is contained in the result set;
- **ORDER BY** clauses, which influence only the order of the result set, which is not validated by PQS;
- aggregate functions, which compute values over multiple rows, when only a single row is present in a table, which allows partially testing them;
- DBMS-specific query options, such as the MySQL-specific `FOR` **UPDATE** clause, which must not influence the result set.

These additional elements are an optional extension to our core approach, and allowed PQS to find additional bugs by stressing the DBMSs' query optimizer. However, they do not comprehensively test these features.

Checking containment. After using the DBMS to evaluate the query in step ⑥, checking whether the pivot row is part of the result set is the last step of our approach. While the checking routine could have been implemented in SQLancer, we instead construct the query so that it checks for containment, effectively combining steps ⑥ and ⑦. DBMSs provide various operators to check for containment, such as the **IN** and **INTERSECT** operators. For example, for checking containment in Figure 1 step ⑦, we can check whether the row `(3, TRUE, -5)` is contained in the result set using the query shown in Listing 2, which returns a row if the pivot row is contained.

Checking arbitrary expressions. An extension of the initial idea of PQS is to use arbitrary expressions to specify which data to fetch in the query of step ⑤, rather than referring to columns only. For example, rather than referring to `t0.c0`, we might want to check whether `t0.c0 + 1` evaluates correctly. To this end, we can generalize the definition of a pivot row to refer to arbitrary computed values. For example, the pivot row value for `t0.c0 + 1` must be `4`, which can be derived based on the expression evaluation mechanism already explained for step ②. In terms of implementation, this thus requires that first the expressions to be used in step ⑤ must be generated, so that they can be evaluated to derive the pivot row values as part of step ②.

3.3 Random State Generation

In step ①, we generate a random database state. Similarly to the generation of queries, we heuristically and iteratively select a number of applicable options. The first step is fixed and consists of creating a number of tables, using the **CREATE TABLE** statement. Subsequent statements are chosen heuristically. Among the applicable options is the **INSERT** statement, which allows inserting data rows. By generating Data Definition Language as well as Data Manipulation Language statements, we can explore a larger space of databases, some of which exposed DBMS bugs. For example, we implemented **UPDATE**, **DELETE**, **ALTER TABLE**, and **CREATE INDEX** commands

Listing 2: Checking containment using the **INTERSECT** operator in SQLite.

```
SELECT (3, TRUE, -5) INTERSECT SELECT t0.c0, t0.c1
    , t1.c0 FROM t1, t2 WHERE NOT(NOT(t0.c1 OR (t1
    .c0 > 3)));
```

for all databases, as well as DBMS-specific run-time options. A number of commands that we implemented were unique to the respective DBMS. Statements unique to MySQL were REPAIR **TABLE** and **CHECK TABLE**. The statements DISCARD and **CREATE** STATISTICS were unique to PostgreSQL. Since the statements are chosen heuristically, the database state generation step might yield an empty database (*e.g.*, because a **DELETE** statement might have deleted all rows, or because a table constraint might make it impossible to insert any rows); in such a case, the current database is discarded and a new database is created. The random database generation is not a contribution of this paper; in fact, many database generation approaches have been proposed, any of which could be paired with PQS [5, 6, 17, 20, 27, 37].

3.4 Important Implementation Details

This section explains implementation decisions, which we consider significant for the outcome of our study.

Error handling. We attempt to generate statements that are correct both syntactically and semantically. However, generating semantically correct statements is sometimes impractical. For example, an **INSERT** might fail when a value already present in a **UNIQUE** column is inserted again; preventing such an error would require scanning every row in the respective table. Rather than checking for such cases, which would involve additional implementation effort and a run-time performance cost, we defined a list of error messages that we might expect when executing the respective statement. Often, we associated an error message to a statement depending on presence or absence of specific keywords; for example, an **INSERT OR** IGNORE is expected to ignore many error messages that would appear without the **OR** IGNORE. If the DBMS returns an expected error, it is ignored. Unexpected errors indicate bugs in the DBMS. For example, in SQLite, a *malformed database disk image* error message is always unexpected, since it indicates the corruption of the database.

Performance. We optimized SQLancer to take advantage of the underlying hardware. We parallelized the system by running each thread on a distinct database, which also resulted in bugs connected to race conditions being found. To fully utilize each CPU, we decreased the probability of SQL statements being generated that cause low CPU utilization (such as VACUUM in PostgreSQL). Typically, SQLancer generates 5,0000 to 20,000 statements per second, depending on the DBMS under test. Since the DBMSs we tested processed queries much faster than other statements, SQLancer generates 100,000 random queries for each database. We implemented the system in Java. However, any other programming language would have been equally well suited, as the performance bottleneck was the DBMS executing the queries.

Number of rows. We found most bugs by restricting the number of rows inserted to a low value (10–30 rows). A higher number would have caused queries to time out when tables are joined without a restrictive join clause. For example, in a query **SELECT** * **FROM** t0, t1, t2, the largest result set for 100 rows in each table would already be $|t0| * |t1| * |t2| = 1,000,000$, significantly lowering the query throughput. A potential concern is that this might prevent PQS from detecting bugs that are triggered only for tables with many rows. We believe that future work could tackle this by generating targeted queries for which the cardinality of the result is bounded.

Database state. For the generation of many SQL statements, knowledge of the database schema or other database state is required; for example, to insert data, SQLancer must determine the name of a table and its columns. We query such state dynamically from the DBMS, rather than tracking or computing it ourselves, which would require additional implementation effort. For example, to query the name of the tables, both MySQL and PostgreSQL provide an information table information_schema.tables and SQLite a table sqlite_master.

Bailouts. For some operators or functions, corner-case behavior (*e.g.*, how an integer operation behaves on an integer overflow) might be difficult to implement, and—at least initially—be less important to test. Unlike the DBMS, the expression evaluation step in our approach is not required to compute a result for every possible input; in our implementation, each operation can bail out during evaluation by throwing an exception, indicating that a new expression should be generated. We also use this mechanism to prevent reporting known bugs, by bailing out when input is encountered that is known to potentially trigger an already-reported bug.

Value caching. When randomly generating values, SQLancer stores values in a cache, which are subsequently re-used with a given probability. Our intuition was that this would more likely trigger interesting corner cases (*e.g.*, when comparing the same values such as 3 > 3). Additionally, we expected this to increase the chance of successfully generating rows for tables that constraint a column to refer to another table (*i.e.*, foreign key constraints).

Implementation scope. Each testing implementation that we realized is extensive, but incomplete. For each DBMS, we implemented at least integer and string data types; for the SQLite implementation, which is the most complete one, we also support floating-point numbers and binary data. We implemented the generation of many common statements, operators, and functions. Given the size of the implementation, exhaustively enumerating all supported features is infeasible; the artifact associated with the paper can be used to investigate which features are supported. Section 5.3 gives an overview of the size of each testing implementation.

4 Evaluation

We evaluated whether the proposed approach is effective in finding bugs in DBMSs. We expected it to detect logic bugs, which cannot be found by fuzzers, rather than crash bugs. This section overviews the experimental setup, bugs found, and characterizes the SQL statements used to trigger the bugs. We then present a DBMS-specific bug overview, where we present interesting bugs and bug trends. To put these findings into context, we measured the size of SQLancer's components and the coverage it reaches on the tested DBMSs.

4.1 Experimental Setup

To test the effectiveness of our approach, we implemented SQLancer and tested SQLite, MySQL, and PostgreSQL in a period of about three months. We conducted all experiments using a laptop with a 6-core Intel i7-8850H CPU at 2.60 GHz and 32 GB of memory running Ubuntu 19.04. Typically, we enhanced SQLancer to test a new operator or DBMS feature, let the tool run for several seconds up to a day, and inspected the bugs found during this process. We automatically reduced test cases to minimal versions [41], and reduced them further manually when this helped to better demonstrate the underlying bug. Finally, we reported any new bugs found during this process. Where possible, we waited for bug fixes before continuing testing and implementing new features.

Baseline. There is no applicable baseline to which we could compare our work. RAGS [46], which was proposed more than 20 years ago, would be the closest related work, but is not publicly available. Due to the small common SQL core, we would expect that RAGS could not find most of the bugs that we found. Khalek et al. worked on automating testing DBMSs using constraint solving [3, 27], with which they found a previously unknown bug. Also their tool is not available publicly. SQLsmith [45], AFL [2] as well as other random query generators and fuzzers [39] only detect crash bugs in DBMSs. Thus, the only potential overlap between these tools and SQLancer would be the crash bugs that we found, which are not the focus of this work.

DBMS versions. For all DBMSs, we started testing the latest release version, which was SQLite 3.28, MySQL 8.0.16, and PostgreSQL 11.4. For SQLite, we switched to the latest trunk version (*i.e.*, the latest non-release version of the source code) after the first bugs were fixed. For MySQL, we also tested version 8.0.17 after it was released. For PostgreSQL, we switched to the latest beta version (PostgreSQL Beta 2) after opening duplicate bug reports. Eventually, we continued to test the latest trunk version.

Bottleneck. We found that duplicate bugs were a significant factor that slowed down our testing. After reporting a bug, we typically waited for bug fixes before continuing our bug-finding efforts; for bugs that were not quickly fixed, we attempted to avoid generating bug-inducing test cases that triggered known bugs. For SQLite, the developers reacted to most of our bug reports shortly after reporting them, and fixed issues typically within a day. Consequently, we focused our testing efforts on this DBMS. For SQLite, we also tested VIEWS, non-default **COLLATE**s (which define how strings are compared), floating-point support, and aggregate functions, which we omitted for the other DBMSs. For MySQL, bug reports were typically verified within a day by a tester. MySQL's development is not open to the general public. Although we tried to establish contact with MySQL developers, we could not obtain any information that went beyond what is visible on the public bug tracker. Thus, it is likely that some of the verified bug reports will subsequently be considered as duplicates or classified to work as intended. Furthermore, although MySQL is available as open-source software, only the code for the latest release version is provided, so any bug fixes could be verified only with the subsequent release. This was a significant factor that restricted us in finding bugs in MySQL; due to the increased effort of verifying whether a newly found bug was already reported, we invested limited effort into testing MySQL. For PostgreSQL, we received feedback to bug reports within a day, and it typically took multiple days or weeks until a bug was fixed, since possible fixes and patches were discussed intensively on the mailing list. As we found fewer bugs for PostgreSQL overall, the response time did not restrict our testing efforts. Note that not all confirmed bugs were fixed. For example, for one reported bug, a developer decided to "*put this on the back burner until we have some consensus how to proceed on that*"; from the discussion, we speculate that the changes needed to address the bug properly were considered too invasive.

Table 2: Total number of reported bugs and their status.

			Closed	
DBMS	Fixed	Verified	Intended	Duplicate
SQLite	64	0	4	2
MySQL	17	7	2	4
PostgreSQL	5	3	7	6

4.2 Bug Reports Overview

Table 2 shows the number of bugs that we reported (121 overall). We considered 96 bugs as true bugs, as they resulted in code fixes (78 reports), documentation fixes (8 reports), or were confirmed by the developers (10 reports). Each such bug was previously unknown and has a unique fix associated with it, or has been confirmed by the developers to be a unique bug. We opened 25 bug reports that we classified as false bugs, because behavior exhibited in the bug reports was considered to work as intended (13 reports) or because bugs that we reported were considered to be duplicates (12 reports).

Table 3: A classification of the true bugs by the bug kind.

DBMS	Logic	Error	SEGFAULT
SQLite	46	16	2
MySQL	14	9	1
PostgreSQL	1	7	0
Sum	61	32	3

Severity levels. Only for SQLite, bugs were assigned a severity level by the DBMS developers. 14 bugs were classified as *Critical*, 8 bugs as *Severe*, and 16 as *Important*. For 13 bugs, we reported them on the mailing list and no entry in the bug tracker was created. The other bug reports were assigned low severity levels such as *Minor*. While the severity level was not set consistently, this still provides evidence that we found many critical bugs.

Bug classification. Table 3 shows a classification of the true bugs. The containment oracle, which found all logic bugs, accounts for most of the bugs that we found, which is expected, since our approach mainly builds on this oracle. Perhaps surprisingly, encountering unexpected errors also allowed us to detect a large number of bugs. For PostgreSQL, we even found 7 unexpected-error bugs, while finding only 1 logic bug. We believe that this observation could be used when using fuzzers to test DBMSs, for example, by checking for specific error messages that indicate database corruptions. Our approach also detected a number of crash bugs, one of which was considered a security vulnerability in MySQL (CVE-2019-2879). These bugs are less interesting, since they could also have been found by traditional fuzzers. In fact, a duplicate bug report was reported for PostgreSQL, based on a SQLsmith finding, shortly after we found and reported it.

4.3 SQL Statements Overview

Test case length. Our automatically and manually reduced test cases—which comprise both the statements used to generate the state, as well as the bug-inducing query—typically comprised only a few SQL statements (3.71 LOC on average). For 13 test cases, a single line was sufficient. Such test cases were either **SELECT** statements that operated on constants, or operations that set DBMS-specific options. The maximum number of statements required to reproduce a bug was 8. A PostgreSQL crash bug that had already been fixed when we reported it required even 27 statements to be reproduced. Overall, the small number of statements required to reproduce a bug suggests that statements and queries could be systematically generated to efficiently, rather than randomly, explore the space (*e.g.*, such as the bounded black-box testing approach implemented in *ACE* [35]).

Statement distribution. Figure 2 shows the distribution of statements. Note that for some bug reports, we had to select the simplest test case among multiple failing ones, which might skew these results. The **CREATE TABLE** and **INSERT** statements are part of most bug reports for all DBMSs, which is expected, since only few bugs can be reproduced without manipulating or fetching data from a table. 91.0% of the bug reports included only a single table. The **SELECT** statement also ranks highly, since the containment oracle relies on it. In all DBMSs, the **CREATE INDEX** statements rank highly; especially for SQLite, we reported a number of bugs where creating an index resulted in a malformed database image or in a row not being fetched. We found that statements that compute or recompute table state were error-prone, for example, REPAIR **TABLE** and **CHECK TABLE** in MySQL, as well as VACUUM and REINDEX in SQLite and PostgreSQL. DBMS-specific options, such as **SET** in MySQL and PostgreSQL, and PRAGMA in SQLite also resulted in bugs being found. For PostgreSQL, some test cases contained ANALYZE, which gathers statistics to be used by the query planner.

Column constraints. Column constraints, which can be used to restrict the values stored in a column, were often part of test cases. The most common constraint was **UNIQUE**(appearing in 21.9% of the test cases). Also **PRIMARY KEY** columns were frequent (16.7%). Typically, the DBMSs enforce **UNIQUE** and **PRIMARY KEY** by creating indexes; explicit indexes, created by **CREATE INDEX** were more common, however (27.1%). Other constraints were uncommon, for example, **FOREIGN KEY**s appeared only in 1.0% of the bug reports.

5 Interesting Bugs

In this section, we present bugs that we found using PQS. We chose bugs that we considered to be interesting, meaning that the selection is necessarily subjective.

5.1 Containment Bugs

We consider bugs found by the containment oracle to be the most interesting, and we designed PQS to specifically find these kind of bugs.

First SQLite bug. Listing 3 shows a test case for the first bug that we found with our approach, and where SQLite failed to fetch a row. The **COLLATE** NOCASE clause instructs the DBMS to ignore the casing when comparing strings; in this test case, it unexpectedly caused the upper-case 'A' to be omitted from the result set. The bug was classified as *Severe* and goes back to when WITHOUT ROWID tables were introduced in 2013. It is a typical bug that we found in SQLite, since it relies on multiple features. As with this bug, 17 of our SQLite bug reports included indexes, 11 included **COLLATE** sequences, and 5 WITHOUT ROWID tables.

SQLite skip-scan optimization bug. A number of SQLite bugs stem from incorrect optimizations, such as the one in

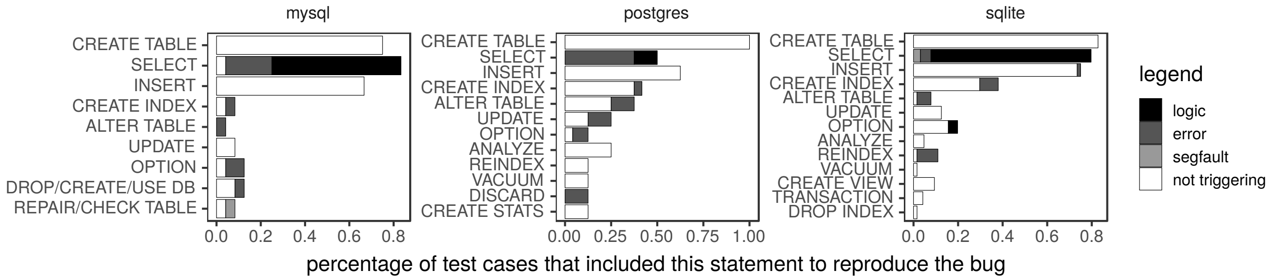


Figure 2: The distribution of the SQL statements used in the bug reports to reproduce the bug. A non-white filling indicates that a statement of the respective category triggered the bug, which was exposed by the test oracle as indicated by the filling (*i.e.*, it was the last statement in the bug report).

Listing 3: The first bug that we found with our approach involved a **COLLATE** index, and a WITHOUT ROWID table.

```
CREATE TABLE t0(c0 TEXT PRIMARY KEY)
       WITHOUT ROWID;
CREATE INDEX i0 ON t0(c0 COLLATE NOCASE);
INSERT INTO t0(c0) VALUES ('A');
INSERT INTO t0(c0) VALUES ('a');
SELECT * FROM t0; -- {'a'}🐞 {'A', 'a'}✓
```

Listing 4: SQLite's skip-scan optimization was implemented incorrectly for **DISTINCT**.

```
CREATE TABLE t0(c0, c1, c2, c3, PRIMARY KEY (c3,
    c2));
INSERT INTO t0(c2) VALUES (0),(0),(0),(0),(0),
  (0),(0),(0),(0),(0),(NULL),(1),(0);
UPDATE t0 SET c1 = 0;
INSERT INTO t0(c0) VALUES (0), (0), (NULL), (0),
    (0);
ANALYZE t0;
UPDATE t0 SET c2 = 1;
SELECT DISTINCT * FROM t0 WHERE c2 = 1; -- {NULL
    |0|1|NULL}🐞 {NULL|0|1|NULL, 0|NULL|1|NULL,
    NULL|NULL|1|NULL}✓
```

Listing 4. For the query in this test case, the skip-scan optimization, where an index is used even if its columns are not part of the **WHERE** clause, was implemented incorrectly for **DISTINCT** queries. The bug was classified as *Severe*.

SQLite unexpected type. Listing 5 shows a bug where an optimization for the **LIKE** operator was implemented incorrectly when applied to **INT** values. The operator was expected to fetch the row, since it checks for an exact string match, but omitted the row from the result set. While this is a minor bug, it is nevertheless interesting, considering that only SQLite allows storing a value of a type that does not match the column declaration. We found this feature to be error-prone, and discovered 8 bugs related to it.

MySQL engine-specific bug. Unlike the other DBMSs we tested, MySQL provides various engines that can be assigned to tables. Listing 6 demonstrates one bug where a row was not fetched when using the MEMORY engine. This was one of 5 bugs that were triggered only when using a non-default engine. This test case is also interesting, as it is one of 4 MySQL test cases that relies on a cast to an unsigned integer, a type that is not provided by the other DBMSs we tested.

Listing 5: We discovered 4 bugs in a **LIKE** optimization, one demonstrated by this test case.

```
CREATE TABLE t0(c0 INT UNIQUE COLLATE NOCASE);
INSERT INTO t0(c0) VALUES ('./');
SELECT * FROM t0 WHERE c0 LIKE './'; -- {}🐞
    {'./'}✓
```

Listing 6: We found 5 bugs using non-default engines in MySQL.

```
CREATE TABLE t0(c0 INT);
CREATE TABLE t1(c0 INT) ENGINE = MEMORY;
INSERT INTO t0(c0) VALUES(0);
INSERT INTO t1(c0) VALUES(-1);
SELECT * FROM t0, t1 WHERE CAST(t1.c0 AS
  UNSIGNED) > IFNULL("u", t0.c0); -- {}🐞 {0|-1}✓
```

Listing 7: Custom comparison operator results in incorrect result.

```
CREATE TABLE t0(c0 TINYINT);
INSERT INTO t0(c0) VALUES(NULL);
SELECT * FROM t0 WHERE NOT(t0.c0 <=> 2035382037);
    -- {}🐞 {NULL}✓
```

MySQL value range bug. We found bugs in MySQL where queries were handled incorrectly depending on the magnitude of an integer or floating-point number. For example, Listing 7 shows a bug where the MySQL-specific <=> inequality operator, which yields a boolean value even when an argument is **NULL**, yielded **FALSE** when the column value was compared with a constant that was greater than what the column's type can represent.

MySQL double negation bug. Listing 8 shows an interesting optimization bug that we found in MySQL. MySQL optimized away the double negation, which appears to be cor-

Listing 8: Double negation bug in MySQL.

```
CREATE TABLE t0(c0 INT);
INSERT INTO t0(c0) VALUES(1);
SELECT * FROM t0 WHERE 123 != (NOT (NOT 123)); --
    {}🐞 {1}✓
```

Listing 9: Table inheritance bug in PostgreSQL.

```
CREATE TABLE t0(c0 INT PRIMARY KEY, c1 INT);
CREATE TABLE t1(c0 INT) INHERITS (t0);
INSERT INTO t0(c0, c1) VALUES(0, 0);
INSERT INTO t1(c0, c1) VALUES(0, 1);
SELECT c0, c1 FROM t0 GROUP BY c0, c1; -- {0|0}🐞
    {0|0, 0|1}✓
```

Listing 10: This bug report caused the SQLite developers to disallow double quotes in indexes.

```
CREATE TABLE t0(c0, c1);
INSERT INTO t0(c0, c1) VALUES ('a', 1);
CREATE INDEX i0 ON t0("C3");
ALTER TABLE t0 RENAME COLUMN c0 TO c3;
SELECT DISTINCT * FROM t0;--{'C3'|1}🐞{'a'|1}✓
```

rect on the first sight. However, since MySQL's flexible type system allows, for example, integers as argument to the **NOT** operator, this optimization is not generally correct. Applying **NOT** to a non-zero integer value should yield 0, and negating 0 should yield 1, which is why the predicate in the **WHERE** clause must yield **TRUE**. However, after optimizing away the double negation, the predicate effectively corresponded to `123 != 123`, which evaluated to **FALSE**, and omitted the pivot row. We considered this case as a duplicate, since the underlying bug that this test case demonstrates seems to have been fixed already in a version not released to the public. We believe that the implicit conversions provided by MySQL (and also SQLite) is one of the reasons that we found more bugs in these DBMSs than in PostgreSQL.

PostgreSQL inheritance bug. In PostgreSQL, we found only one logic bug. The bug was related to table inheritance, a feature that only PostgreSQL provides (see Listing 9). Table `t1` inherits from `t0`, and PostgreSQL merges the `c0` column in both tables. As described in the PostgreSQL documentation, `t1` does not respect the **PRIMARY KEY** restriction of `t0`. This was not considered when implementing the **GROUP BY** clause, which caused PostgreSQL to omit one row in its result set.

SQLite double quote bug. Listing 10 shows a test case, for which, after the `RENAME` operation, it is ambiguous whether the index refers to a string or column. The **SELECT** fetches `C3` as a value for the column `c3`, which is incorrect in either case. SQLite allowed both single quotes and double quotes to be used to denote strings; depending on the context, either can refer to a column name. After we reported the bug, a breaking change that disallowed strings in double quotes when creating indexes was introduced.

Listing 11: We found 4 malformed database errors in SQLite using the error oracle, such as this one.

```
CREATE TABLE t1 (c0, c1 REAL PRIMARY KEY);
INSERT INTO t1(c0, c1) VALUES (TRUE,
    9223372036854775807), (TRUE, 0);
UPDATE t1 SET c0 = NULL;
UPDATE OR REPLACE t1 SET c1 = 1;
SELECT DISTINCT * FROM t1 WHERE c0 IS NULL;--
    Error: database disk image is malformed 🐞
```

Listing 12: Unexpected null value bug in PostgreSQL.

```
CREATE TABLE t0(c0 TEXT);
INSERT INTO t0(c0) VALUES('b'), ('a');
ANALYZE;
INSERT INTO t0(c0) VALUES (NULL);
UPDATE t0 SET c0 = 'a';
CREATE INDEX i0 ON t0(c0);
SELECT * FROM t0 WHERE 'baaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaa' > t0.c0; -- Error: found
    unexpected null value in index "i0"🐞
```

5.2 Error Bugs

While finding error bugs was not the main goal of our work, they were common, which is why we discuss two such cases.

SQLite database corruption. Listing 11 shows a test case where manipulating values in a `REAL` **PRIMARY KEY** column resulted in a corrupted database. We found 4 such cases, as indicated by *malformed database schema* errors. This specific bug was introduced in 2015, and went undetected until we reported it in 2019; it was assigned a *Severe* severity level.

PostgreSQL multithreaded error. Listing 12 shows a bug that was triggered only when another thread opened a transaction, holding a snapshot with the **NULL** value. In order to reproduce such bugs, we had to record and replay traces of all executing threads. 4 reported PostgreSQL bugs (including closed/duplicate ones) could be reproduced only when running multiple threads.

5.3 Implementation Size and Coverage

Implementation effort. It is difficult to quantify the effort that we invested in implementing support for each DBMS, since, for example, we got more efficient in implementing support over time. The LOC of code of the individual testing components (see Table 4) reflects our estimates that we invested the most effort to test SQLite, then PostgreSQL, and then MySQL. The code part shared by the components is rather small (918 LOC), which provides evidence for the different SQL dialects that they support. We believe that the implementation effort for SQLancer is small when compared to the size of the tested DBMSs. The LOC in this table were derived after compiling the respective DBMS using default configurations, and thus include only those lines reachable in the binary. Thus, they are significantly smaller than the ones we derived statically for the entire repositories in Table 1.

Table 4: The size of SQLancer's components specific and common to the tested databases.

	LOC			Coverage	
DBMS	SQLancer	DBMS	Ratio	Line	Branch
SQLite	6,501	49,703	13.1%	43.0%	38.4%
MySQL	3,995	707,803	0.6%	24.4%	13.0%
PostgreSQL	4,981	329,999	1.5%	23.7%	16.6%

Coverage. To estimate how much code of the DBMSs we tested, we instrumented each DBMS and ran SQLancer for 24 hours (see Table 4). The coverage appears to be low (less than 50% for all DBMSs); however, this is expected, because we were only concerned about testing data-centric SQL statements. MySQL and PostgreSQL provide features such as user management, replication, and maintenance functionalities, which we did not test. Furthermore, all DBMSs provide consoles to interact with the DBMS and programming APIs. We currently do not test many data types, language elements such transaction savepoints, many DBMS-specific functions, configuration options, and operations that might conflict with other threads running on a distinct database. The coverage for SQLite is the highest, reflecting that we invested the most effort in testing it, but also that it provides fewer features in addition to its SQL implementation.

6 Discussion

Number of bugs and code quality. The number of bugs that we found in the respective DBMSs depended on many, difficult-to-quantify factors. We found most bugs in SQLite. A significant reason for this is that we focused on this DBMS, because the developers quickly fixed all bugs. Furthermore, while the SQL dialect supported by SQLite is compact, we perceived it to be the most flexible one; for example, column types are not enforced, leading to bugs that were not present in PostgreSQL, and to a lesser degree in MySQL. MySQL's release policy made it difficult to test it efficiently, limiting the number of bugs that we found in this DBMS. In PostgreSQL, we found the least number of bugs, and we believe that a significant reason for this is that the SQL dialect support is strict, and few implicit conversions are performed.

False positives. In principle, PQS does not report false positives; that is, bugs found by PQS are always real bugs. Nevertheless, false positives can be due to a limited understanding of the DBMS operator's expected behavior when implementing the operator's `execute()` method. Consequently, the 13 bug reports that were considered to work as intended were either due to (1) an incorrect implementation of an operator in PQS, or (2) a bug found by the *error* oracle where the error was expected. False bug reports allowed us to refine our implementation based on the DBMS developer's feedback. In 8 cases, bug reports also led to documentation enhancements or fixes.

Common bugs. Common bugs that we found among all DBMSs were optimization bugs (*i.e.* where a performance optimization caused correctness issues). Often, these were related to indexes created either explicitly (*i.e.* using **`CREATE INDEX`**) or implicitly (*e.g.*, using a **`UNIQUE`** constraint), as described in Section 4.3. A number of bugs were related to the handling of **`NULL`**, which seems to be difficult to reason about for DBMS developers. Most of the other bugs we found were unique to the respective DBMS.

Existing test efforts. All three DBMSs are extensively tested. For example, SQLite, for which we found most bugs, has 662 times as much test code and test scripts than source code [47]. The core is tested by three separate test harnesses. The TCL tests comprise 45K test cases, the TH3 proprietary test harness contains about 1.7 million test instances and provides 100% branch test coverage and 100% MC/DC test coverage [25], and the SQL Logic Test runs about 7.2 million queries based on over 1 GB of test data. SQLite uses various fuzzers such as a random query generator called *SQL Fuzz*, a proprietary fuzzer *dbsqlfuzz*, and it is fuzzed by Google's OSS Fuzz project [14]. Other kinds of tests are also applied, such as crash testing, to demonstrate that the database will not go corrupt on system crashes or power failures. Considering that SQLite and other DBMSs are tested this extensively, we believe that it is surprising that SQLancer could find any bugs.

Deployment. One question is how DBMS developers would use PQS during development. Similarly to fuzzers, dynamic testing approaches like PQS cannot provide any guarantees in terms of bug-finding outcomes. Consequently, it is also unclear on how long SQLancer should be run to find all bugs it would be able to find. In practice, it might be useful to run SQLancer similarly to fuzzers, for example, either for a limited period as part of an overnight continuous integration process, or constantly to maximize the chances of finding new bugs. Future work might investigate the systematic enumeration of queries, while also pruning the infinitely large space of possible queries, to give bounded guarantees.

Specification. In order to implement the expression evaluation, we implemented AST interpreters that evaluate the operators based on the pivot row. This evaluation step essentially encodes the specification against which the DBMS is checked. We implemented the expression evaluation primarily based on each DBMS' documentation. Where we deemed the documentation to be insufficient, we used a trial-and-error approach to implement the correct semantics. In contrast to differential testing, where a difference in the semantics between two DBMSs' SQL dialect would result in repeated false positives, diverging behavior in an implementation of PQS (*e.g.*, caused by implementation errors) can be addressed by code fixes. In fact, this observation can be used to effectively test the PQS implementation, by running it against the DBMS under test, rather than—or in addition to—using manually-written unit tests.

Limitations. PQS has a number of limitations in terms of what logic bugs it can find. PQS only partly validates a query's result, and thus, in general, is inapplicable to, for example, check the correct insertion or deletion of records, detect concurrency bugs, bugs related to transactions, or bugs in the access control layer of DBMSs [19]. Conceptually, PQS cannot detect duplicate rows that are mistakenly omitted from or included in the result set, since duplicate records are indistinguishable for PQS. Consequently, it also cannot be used to validate the cardinality of a result set, even when each of its rows is once selected as a pivot row. PQS is not suited for testing the OFFSET and **LIMIT** clauses, since they might exclude the pivot row from the result set. Although PQS has found 3 bugs in aggregate functions, it can only do so in corner cases, such as when aggregate functions are used in a view that is queried, or when a table contains only a single row, in which case the result of the aggregate function can be determined easily. Similarly, PQS cannot find bugs in window functions, which also compute their result over multiple rows in a window. While SQLancer generates **ORDER BY** clauses, PQS cannot validate the result set's ordering. Similarly, for **GROUP BY** clauses, PQS cannot confirm that all duplicate values are grouped. PQS cannot be used to test **NOT EXISTS** predicates that reference tables (*i.e.*, semi-joins), since the approach cannot ensure that a row is not contained based on only the pivot row. Similarly, while PQS can be used to test joins, it can only test for combinations where a **JOIN** clauses matches rows on both the left and right side of a join; for example, for a **LEFT JOIN**, it is inapplicable to test cases where only values for the left table are fetched, but not the right one. PQS is unable to test the results of ambiguous queries and queries that rely on nondeterministic functions (such as used to generate random numbers), since it is based on the assumption that the result set is unambiguous. It is also unable to test user-provided functions or operators, unless they are re-implemented in PQS. Supporting these makes interesting future work. PQS, as the first practical technique for finding logic bugs in DBMSs, has demonstrated its effectiveness by finding a wide variety of bugs such as in operator implementations and optimizations.

Implementation effort. Since the supported SQL dialects differ vastly between DBMSs, we had to implement DBMS-specific components in SQLancer. It could be argued that the implementation effort is too high, especially when the full support of a SQL dialect is to be tested, which could arguably be similar to implementing a new DBMS. Indeed, we could not test complex functions such as SQLite's printf, which would have required significant implementation effort. However, we still argue that the implementation effort is reasonably low, and allows testing significant parts of a DBMS. Specificially, based on our experiments, implementing sargable predicates (*e.g.* those predicates for which the DBMS can use an index), already allows finding the majority of optimization bugs. Furthermore, our approach effectively evaluates only literal expressions, and does not need to consider multiple rows. This obviates the need of implementing a query planner, which typically is the most complex component of a DBMS [13]. Furthermore, the performance of the evaluation engine is insignificant; the performance bottleneck was the DBMS evaluating the queries, rather than SQLancer. Thus, we also did not implement any optimizations, which typically require much implementation effort in DBMSs [15]. Finally, we did not need to consider aspects such as concurrency and multi-user control as well as integrity [53].

7 Related Work

Testing of software systems. This paper fits into the stream of testing approaches for important software systems. Differential testing [33] is a technique that compares the results obtained by multiple systems that implement a common language; if results deviate, one or multiple of the systems are likely to have a bug. It has been used as a basis for many approaches, for example, to test C/C++ compilers [51, 52], symbolic execution engines [24], and PDF readers [30]. Metamorphic testing [9], where the program is transformed so that the same result as for the original program is expected, has been applied to various systems; for example, *equivalence modulo inputs* is a metamorphic-testing-based approach that has been used to find over one thousand bugs in widely-used compilers [31]. As another example, metamorphic testing has been successfully applied to test graphic shader compilers [12]. We present PQS as a novel approach to testing DBMSs, which solves the *oracle problem* in a novel way, namely by checking whether a DBMS works correctly for a specific query and row. We believe that our approach can also be extended to test other software systems that have an internal state, of which a single instance can be selected.

Metamorphic testing of DBMSs. PQS inspired two follow-up testing approaches, namely Non-Optimizing Reference Engine Construction (NoREC) [42] and Ternary Logic Partitioning (TLP) [43], both of which were implemented in SQLancer. Conceptually, NoREC translates a query that is potentially optimized by the DBMS (called the *optimized query*) to a query that cannot effectively be optimized, thus detecting *optimization bugs*—which are a subcategory of logic bugs—when the two query's result sets differ. TLP translates a given query to multiple so-called *partitioning queries*, each of which computes a part of the result, whose combined result is then compared with the given query's result sets. Both are metamorphic testing approaches. Thus, the effort required for implementing them is negligible; however, they cannot establish a ground truth, which PQS can. NoREC could find only 52.7% of the bugs detected by PQS, which is expected due to its narrower scope [42]. Considering that our PQS implementation could also check for non-containment, which is a straightforward implementation enhancement, it could have detected 82.4% of the NoREC bugs. The remaining bugs found only by NoREC are due to bugs in the implementation of the `SUM()` and `COUNT()` aggregate functions, which NoREC uses for a more efficient implementation of the test oracle; it does not provide testing support for aggregates in general.

Differential testing of DBMSs. Slutz proposed an approach *RAGS* for finding bugs in DBMSs based on differential testing [46]. In *RAGS*, queries are automatically generated and evaluated by multiple DBMSs. If the results are inconsistent, a bug has been found. As acknowledged by the author, the approach was very effective, but applies to only a small set of common SQL statements. In particular, the differences in `NULL` handling, character handling, and numeric type coercions were mentioned as problematic. Our approach can detect bugs also in SQL statements unique to a DBMS, but requires separate implementations for each DBMS.

Database fuzzing. SQLsmith is a popular tool that randomly generates SQL queries to test various DBMSs [45]. SQLsmith has been highly successful and has found over 100 bugs in popular DBMSs such as PostgreSQL, SQLite and MonetDB since 2015. However, it cannot find logic bugs found by our approach. Similarly, general-purpose fuzzers such as AFL [2] are routinely applied to DBMSs, and have found many bugs, but also cannot detect logic bugs.

Consistency checking. Kingsbury has developed Jepsen, a framework to test safety properties of distributed systems (such as violations of consistency models), which found many critical bugs in distributed DBMSs [28]. As part of Jepsen, Kingsbury et al. proposed Elle [29], which is a transactional consistency checker. In contrast to PQS, Jepsen aims to find logic bugs primarily in the transaction processing of a DBMS.

Queries satisfying constraints. Some approaches improved upon random query generation by generating queries that satisfy certain constraints, such as cardinalities or coverage characteristics. The problem of generating a query, whose subexpressions must satisfy certain constraints, has been extensively studied [7, 34]; since this problem is complex, it is typically tackled by an approximate algorithm [7, 34]. An alternative approach was proposed by Bati et al. where queries are selected and mutated based on whether they increase the coverage of rarely executed code paths [4], increasing the coverage of the DBMS component under test. Rather than improved query generation, Lo et al. proposed an approach where a database is generated based on specific requirements on test queries [32]. While these approaches improve the query and database generation, they do not help in automatically finding errors, since they do not propose an approach to automatically verify the queries' results.

DBMS testing based on constraint solving. Khalek et al. worked on automating testing DBMSs using constraint solving [3, 27]. Their core idea was to use a SAT-based solver to automatically generate database data, queries, and a test oracle. In their first work, they described how to generate query-specific data to populate a database and enumerate the rows that would be fetched to construct a test oracle [27]. They could reproduce previously-reported and injected bugs, but discovered only one new bug. In follow-up work, they also demonstrated how the SAT-based approach can be used to automatically generate queries [3]. As with our approach, they provide a test oracle, and additionally a targeted data generation approach. While both approaches found bugs, our approach found many previously undiscovered bugs. Furthermore, we believe that the simplicity of our approach could make it wider applicable.

Testing other aspects. Rather than trying to improve the correctness of DBMSs, several approaches were proposed to test other aspects of DBMSs. Poess et. al proposed a template-based approach to generating queries suitable to benchmark DBMSs, which they implemented in a tool QGEN [39]. Similarly to random query generators, QGEN could also be used to test DBMSs. Gu et al presented an approach to quantify an optimizer's accuracy for a given workload by defining a metric over different execution plans for this workload, which were generated by using DBMS-specific tuning options [18]. Jung et al. found performance bugs based on several versions of a given DBMS [23]. Zheng et al. tested the ACID properties provided by the DBMS in the presence of power faults [53]. These approaches, however, cannot be used to find logic bugs.

8 Conclusion

We have presented an effective approach for detecting bugs in DBMSs, which we implemented in a tool SQLancer, with which we found over 96 bugs in three popular and widely-used DBMSs. The effectiveness of SQLancer is surprising, considering the simplicity of our approach, and that we only implemented a small subset of features that current DBMSs support. There are a number of promising directions that could help uncover additional bugs or improve PQS otherwise, which we regard as future work. SQLancer generates tables with a low number of rows to prevent timeouts of queries when multiple tables are joined with non-restrictive conditions. By generating targeted queries with conditions based on table cardinalities [7, 34], we could test the DBMSs for a large number of rows, better stressing the query planner [13]. A disadvantage of PQS is that it needs to be implemented for every DBMS to be tested. As part of future work, this effort could be reduced, for example, by providing common building blocks that could be combined to implement operators and functions more efficiently. Finally, PQS could be extended to also test for rows that are incorrectly fetched by selecting a pivot row, ensuring that the randomly-generated predicates evaluate to `FALSE` or `NULL` for it, and then check that the pivot row is not contained in the result set.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Yuanyuan Zhou, for their insightful feedback. We want to thank all the DBMS developers for responding to our bug reports as well as analyzing and fixing the bugs we reported. We especially want to thank the SQLite developers, D. Richard Hipp and Dan Kennedy, for taking all bugs we reported seriously and fixing them quickly. Furthermore, we are grateful for the feedback received by our colleagues at ETH Zurich.

References

[1] DB-Engines Ranking (December 2019), 2019. https://db-engines.com/en/ranking.

[2] american fuzzy lop, 2020. https://github.com/google/AFL.

[3] Shadi Abdul Khalek and Sarfraz Khurshid. Automated sql query generation for systematic testing of database engines. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 329–332, 2010.

[4] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. A genetic approach for random testing of database systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1243–1251. VLDB Endowment, 2007.

[5] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. Qagen: Generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, page 341–352, New York, NY, USA, 2007. Association for Computing Machinery.

[6] Nicolas Bruno and Surajit Chaudhuri. Flexible database generators. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, page 1097–1107. VLDB Endowment, 2005.

[7] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE Trans. on Knowl. and Data Eng.*, 18(12):1721–1725, December 2006.

[8] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pages 249–264, 1974.

[9] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong, 1998.

[10] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[11] E.F. Codd. *Relational Completeness of Data Base Sublanguages*. Research report // San José Research Laboratory: Computer sciences. IBM Corporation, 1972.

[12] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.*, 1(OOPSLA):93:1–93:29, October 2017.

[13] Leo Giakoumakis and César A Galindo-Legaria. Testing sql server's query optimizer: Challenges, techniques and experiences. *IEEE Data Eng. Bull.*, 31(1):36–43, 2008.

[14] Google. Announcing oss-fuzz: Continuous fuzzing for open source software, 2016. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html.

[15] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[16] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, page 209–218, USA, 1993. IEEE Computer Society.

[17] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. *SIGMOD Rec.*, 23(2):243–252, May 1994.

[18] Zhongxian Gu, Mohamed A. Soliman, and Florian M. Waas. Testing the accuracy of query optimizers. In *Proceedings of the Fifth International Workshop on Testing Database Systems*, DBTest '12, pages 11:1–11:6, 2012.

[19] Marco Guarnieri, Srdjan Marinovic, and David Basin. Strong and provably secure database access control. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy*, pages 163–178. IEEE, 2016.

[20] Kenneth Houkjær, Kristian Torp, and Rico Wind. Simple and realistic data generation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 1243–1246. VLDB Endowment, 2006.

[21] William E. Howden. Theoretical and empirical studies of program testing. In *Proceedings of the 3rd International Conference on Software Engineering*, ICSE '78, pages 305–311, Piscataway, NJ, USA, 1978. IEEE Press.

[22] Matt Jibson. SQLsmith: Randomized sql testing in cockroachdb, 2019. https://www.cockroachlabs.com/blog/sqlsmith-randomized-sql-testing/.

[23] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proc. VLDB Endow.*, 13(1):57–70, September 2019.

[24] Timotej Kapus and Cristian Cadar. Automatic testing of symbolic execution engines via program generation and differential testing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 590–600, Piscataway, NJ, USA, 2017. IEEE Press.

[25] Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. A practical tutorial on modified condition/decision coverage. Technical report, 2001.

[26] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, September 2018.

[27] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. Query-aware test generation using a relational constraint solver. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 238–247, Washington, DC, USA, 2008. IEEE Computer Society.

[28] Kyle Kingsbury. Jepsen, 2020. https://github.com/jepsen-io/jepsen.

[29] Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations, 2020.

[30] Tomasz Kuchta, Thibaud Lutellier, Edmund Wong, Lin Tan, and Cristian Cadar. On the correctness of electronic documents: Studying, finding, and localizing inconsistency bugs in pdf readers and files. *Empirical Softw. Engg.*, 23(6):3187–3220, December 2018.

[31] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 216–226, 2014.

[32] Eric Lo, Carsten Binnig, Donald Kossmann, M. Tamer Özsu, and Wing-Kai Hon. A framework for testing dbms features. *The VLDB Journal*, 19(2):203–230, Apr 2010.

[33] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[34] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. Generating targeted queries for database testing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 499–510, 2008.

[35] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 33–50, Carlsbad, CA, October 2018. USENIX Association.

[36] MySQL. Mysql homepage, 2020. https://www.mysql.com/.

[37] Andrea Neufeld, Guido Moerkotte, and Peter C. Lockemann. Generating consistent test data: Restricting the search space by a generator formula. *The VLDB Journal*, 2(2):173–214, April 1993.

[38] Stack Overflow. Developer survey results 2019, 2019.

[39] Meikel Poess and John M. Stephens, Jr. Generating thousand benchmark queries in seconds. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 1045–1053. VLDB Endowment, 2004.

[40] PostgreSQL. Postgresql homepage, 2019.

[41] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. page 335–346, 2012.

[42] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, 2020.

[43] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020.

[44] Manuel Rigger and Zhendong Su. OSDI 20 Artifact for "Testing Database Engines via Pivoted Query Synthesis", 2020. https://doi.org/10.5281/zenodo.4005704.

[45] Andreas Seltenreich. SQLSmith, 2020. https://github.com/anse1/sqlsmith.

[46] Donald R Slutz. Massive stochastic testing of sql. In *VLDB*, volume 98, pages 618–622, 1998.

[47] SQLite. How SQLite is tested, 2020. https://www.sqlite.org/testing.html.

[48] SQLite. Most widely deployed and used database engine, 2020. https://www.sqlite.org/mostdeployed.html.

[49] SQLite. SQLite homepage, 2020. https://www.sqlite.org/.

[50] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, 2013.

[51] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, 2011.

[52] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 347–361, 2017.

[53] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W Zhao, and Shashank Singh. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 449–464, Broomfield, CO, October 2014. USENIX Association.