Detecting Optimization Bugs in Database Engines via Non-optimizing Reference Engine Construction

Manuel Rigger
manuel.rigger@inf.ethz.ch
Department of Computer Science, ETH Zurich
Zurich, Switzerland

Zhendong Su
zhendong.su@inf.ethz.ch
Department of Computer Science, ETH Zurich
Zurich, Switzerland

ABSTRACT

Database Management Systems (DBMS) are used ubiquitously. To efficiently access data, they apply sophisticated optimizations. Incorrect optimizations can result in *logic bugs*, which cause a query to compute an incorrect result set. We propose *Non-optimizing Reference Engine Construction* (NoREC), a fully-automatic approach to detect optimization bugs in DBMS. Conceptually, this approach aims to evaluate a query by an optimizing and a non-optimizing version of a DBMS, to then detect differences in their returned result set, which would indicate a bug in the DBMS. Obtaining a non-optimizing version of a DBMS is challenging, because DBMS typically provide limited control over optimizations. Our core insight is that a given, potentially randomly-generated *optimized query* can be rewritten to one that the DBMS cannot optimize. Evaluating this unoptimized query effectively corresponds to a non-optimizing reference engine executing the original query. We evaluated NoREC in an extensive testing campaign on four widely-used DBMS, namely PostgreSQL, MariaDB, SQLite, and CockroachDB. We found 159 previously unknown bugs in the latest versions of these systems, 141 of which have been fixed by the developers. Of these, 51 were optimization bugs, while the remaining were error and crash bugs. Our results suggest that NoREC is effective, general and requires little implementation effort, which makes the technique widely applicable in practice.

CCS CONCEPTS

• **Information systems** → **Database query processing**; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

database testing, DBMS testing, query optimizer bugs, test oracle

ACM Reference Format:
Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-optimizing Reference Engine Construction. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368089.3409710

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409710

1 INTRODUCTION

Database Management Systems (DBMS) are an important component in many systems. To meet the growing performance demands, increasingly sophisticated optimizations for query evaluation are applied [18, 35, 39, 60]. Unsurprisingly, the *query optimizer* is typically considered to be a DBMS' most complex component, posing a major correctness challenge [20, 21]. Implementation errors in the optimizer can result in *logic bugs*, which are bugs that cause a DBMS to return an incorrect result set for a given query. Specifically, we refer to logic bugs in the query optimizer as *optimization bugs*. Pivoted Query Synthesis (PQS) was recently proposed as a way of tackling logic bugs in DBMS [46]. Its core idea is to verify the DBMS based on a single *pivot row*, for which a query is generated that is expected to fetch this row. While PQS has been effective in detecting many bugs in widely-used DBMS, a significant drawback is the high implementation effort that is required to realize this technique; specifically, the technique requires the re-implementation of the DBMS' provided operators and functions to determine whether a randomly-generated expression evaluates to `TRUE`. Since PQS considers only a single row, it also fails to detect bugs such as when a duplicate row is mistakenly fetched or omitted. Another successful technique for detecting logic bugs in DBMS was realized in a system called RAGS [50]. It is based on differential testing [36]. A query is generated that is sent to multiple DBMS; if the DBMS disagree on the output, at least one of the DBMS is expected to be affected by a bug. As noted by the authors, a significant drawback of this technique is that it applies only to the common core of SQL, which is small, because DBMS differ in what operators and types they support and because even common operators have subtly different semantics between different DBMS [50].

In this paper, we propose *Non-optimizing Reference Engine Construction (NoREC)*, a novel, general, and cost-effective technique for finding optimization bugs in DBMS. The high-level idea of our approach is to compare the results of an optimizing version of a DBMS against a version of the same DBMS that does not perform any optimizations. Obtaining such a non-optimizing version of a DBMS is challenging. While many DBMS provide some options to control optimizations, these are limited and specific to a DBMS. Although adding such options would be a possibility, doing so retrospectively would be error prone and impractical because of the high implementation effort and domain knowledge required. Rather, we propose the idea that a given query can be rewritten so that the DBMS is not expected to optimize it. Finding a translation mechanism that guarantees the same result as the original query, while making optimizations inapplicable, is not obvious. Our key insight is that this can be achieved by transforming a query with a `WHERE` clause, which is subject to extensive optimization by the DBMS

Listing 1: Illustrative example where a bug in SQLite's *LIKE* optimization caused a record to mistakenly be omitted.

```
CREATE TABLE t0(c0 UNIQUE);
INSERT INTO t0 VALUES (-1);
① SELECT * FROM t0 WHERE t0.c0 GLOB '-*'; -- {}
② SELECT t0.c0 GLOB '-*' FROM t0; -- {TRUE}
```

and the basis for creating an efficient query plan, to a query that evaluates the WHERE clause's predicate on every record of the table, which cannot be meaningfully optimized; the number of records fetched by the first query must be equal to the number of times the WHERE predicate evaluates to TRUE for the second query. A different result indicates a bug in the DBMS.

Listing 1 illustrates the idea of our approach based on a bug that we found in SQLite where an optimization caused a row to be erroneously omitted from the result set. Starting from an initial database that contains a single record, we generate query ① with a random WHERE condition `t0.c0 GLOB '-*'`. GLOB is a regular expression operator, and `'-*'` a regular expression that should match a `'-'`, followed by any number of characters. Since WHERE (and JOIN) clauses are performance-critical, they are subject to optimization by the DBMS. In this example, SQLite applies the *LIKE optimization* [54] by using an index—which is an auxiliary data structure used for efficient lookups and implicitly created based on the UNIQUE constraint—to do a range search, allowing the execution engine to skip irrelevant records. Unexpectedly, the optimization causes the DBMS to omit fetching the single record, even though it matches the specified regular expression. Next, we translate the first query to query ② so that the DBMS is unlikely to optimize it, namely by moving the WHERE clause's predicate directly next to the SELECT keyword, which causes the query to evaluate the predicate on each record of the table. We expect that the number of times the expression evaluates to TRUE corresponds to the actual number of records fetched by the first query. However, in this example, the expression evaluates to TRUE for the single record in the database. The DBMS could only meaningfully apply the incorrect optimization to the first query, but not to the second. We reported this bug to the SQLite developers, who quickly fixed it.

We implemented NoREC in a tool called *SQLancer*, which is available at https://github.com/sqlancer.[1] To demonstrate the generality of our approach, we evaluated NoREC on four widely-used, production-level DBMS, SQLite, MariaDB, PostgreSQL, and CockroachDB. As part of an extensive 5-month testing campaign, in which we sought to demonstrate the effectiveness of the approach and maximize its real-world impact, we found 159 *previously-unknown* bugs, many of which were serious, of which 141 were subsequently fixed and 14 confirmed. These comprised 51 optimization bugs, 23 crash bugs, 27 assertion failures, and 58 error bugs. Although SQLite has been extensively tested by PQS, NoREC found more than 100 additional bugs in it, demonstrating NoREC's effectiveness. The DBMS developers greatly appreciated our efforts. For example, the SQLite website describes our successful testing campaign [52] and mentions the following: "*Rigger's work is currently unpublished. When it is released, it could be as influential as Zalewski's invention of AFL and profile-guided fuzzing.*" We believe that the simplicity, effectiveness, and low implementation effort of NoREC will result in its broad adoption. In summary, this paper contributes the following:

- a new, effective testing technique for DBMS based on a novel test oracle for detecting optimization bugs called NoREC;
- an implementation of NoREC in a tool called SQLancer;
- an extensive evaluation of NoREC, which uncovered more than 150 new bugs in widely-used DBMS.

[1] An artifact prepared for long-term archival is also available [45].

2 BACKGROUND

Database management systems and SQL. DBMS are based on a *data model*, which abstractly describes how data is organized. Most widely-used DBMS are based on the *relational data model* proposed by Codd [14]—according to the DB-Engines Ranking [17], seven of the ten most popular DBMS are based on it. In our work, we primarily aim to test such relational DBMS. Structured Query Language (SQL) [10], which is based on relational algebra [15], is the most commonly used language in relational DBMS to create databases, tables, insert rows, as well as manipulate and retrieve data. Our approach is not directly applicable to NoSQL DBMS, as they often provide their own query languages or support only a SQL subset; however, it is applicable to the newer generation of NewSQL DBMS, which attempt to achieve the same scalability of NoSQL DBMS, but provide SQL as a query language [41].

Automatic testing. In this work, we focus on automatic testing, which is is an effective and practical way of finding bugs, although it cannot guarantee their absence [27]. Two components are crucial for an automatic testing approach. First, an effective test case must stress significant portions of the system under test, to find bugs in them. Second, a *test oracle* is required that detects whether a certain test case executes as expected. While various database generators [4, 7, 23, 26, 32, 38] and query generators [2, 9, 30, 37, 42, 49, 58] have been proposed to generate effective test cases, test oracles have received less attention. As part of this work, we propose an effective, cost-effective test oracle that allows detecting logic bugs in DBMS.

Optimizations in DBMS. Decades of work have been devoted to query optimization [19, 22]. Each DBMS typically provides a query optimizer that inspects a query, potentially simplifies it, and maps it efficiently to physical accesses (*i.e.*, by selecting one of potentially multiple available access paths [48]). Consider the two queries in Listing 1. It is well understood that the primary performance gains of query optimizations stem from determining how the database records can be efficiently fetched. Consequently, the query optimizer would focus its optimization effort on simplifying and creating an efficient query plan based on the WHERE clause in query ①. In query ②, the predicate is evaluated once for every row in the result set, and thus provides limited space for meaningful optimization. As detailed below, we utilize this observation to translate an optimized query to one that is less optimized.

Differential testing. Differential testing [36] refers to a testing technique where a single input is passed to multiple systems that are expected to produce the same output; if the systems disagree on the output, a bug in at least one of the systems has been detected. Slutz applied this technique for testing DBMS in a system called

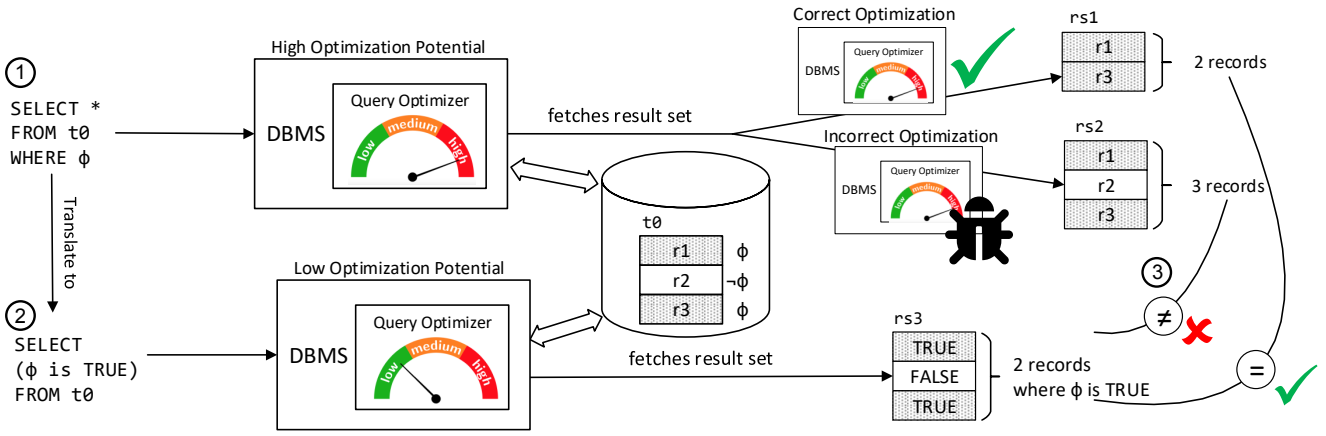

Figure 1: The core of the approach is the translation of an optimized query (step ①) to an unoptimized one (step ②), which allows the automatic detection of optimization bugs (step ③). `t0` is a table contained in the database, and `rs1`, `rs2`, as well as `rs3` are result sets returned by the DBMS. Predicate ϕ is random, but fixed.

RAGS by generating SQL queries that are sent to multiple DBMS and then observing differences in the output sets [50]. While the approach was effective, the author stated that the small common core and the differences between different DBMS were a challenge. Indeed, DBMS typically differ in the SQL dialect that they support, by deviating from the standard and providing DBMS-specific extensions [46]. For example, the CockroachDB developers argued that they cannot use differential testing using PostgreSQL [29], which is the DBMS that is closest to it: *Correctness is difficult because we don't have any oracle of truth, which would require a known working SQL engine, which is exactly the thing we're trying to break. We 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 and doesn't allow us to use the full CockroachDB grammar.* In this paper, we propose an approach that allows building such a "known working SQL engine", namely one that is expected to be free of optimization bugs. As argued next, it is unclear how differential testing could be used to achieve this.

Controlling optimizations in DBMS. One obvious, but infeasible approach to finding optimization bugs is to realize differential testing by executing a SQL query once by disabling, and once by enabling optimizations in a DBMS to detect bug-induced deviations in the result set. This technique has been applied on compilers [33, 61], where programs were compiled without and with optimization flags. For DBMS, the majority of optimizations cannot be disabled. DBMS typically provide *some* run-time and compile-time options to control the behavior of operators and optimizations, but these are typically very limited. For example, the `LIKE` optimization applied to the query in Listing 1 cannot be disabled; SQLite provides only an option to control whether the operator should ignore the casing of the string. Similarly, some DBMS allow the specification of hints to the query optimizer for a given query [8], which also does not apply to many optimizations. Although modifying the DBMS to provide configuration options for all optimizations would be a possibility, doing so would require DBMS-specific knowledge and would involve a high implementation effort.

3 APPROACH

To find optimization bugs in DBMS, we propose NoREC. Our core insight is that a given query that is optimized by the DBMS can be transformed to another query that cannot be effectively optimized. For brevity, we refer to the query that is potentially optimized by the DBMS as the *optimized query*, and the query that is not or less optimized as the *unoptimized query*. While our translation step cannot guarantee the absence of optimizations, we found that this technique is widely applicable to disable them in practice.

3.1 Approach Overview

Figure 1 illustrates our approach. In step ①, we randomly generate an optimized query of the form `SELECT * FROM t0 WHERE` ϕ. Since most optimizations apply to data filtering, such as expressed in the query's `WHERE` clause, we expect that the randomly-generated query might be optimized by the DBMS. In the figure, the database contains a single table `t0` holding the records `r1`, `r2`, and `r3`. Assuming that the DBMS functions correctly, the result set should correspond to `rs1`, which comprises two records `r1` and `r3`. Due to an incorrect optimization, however, it might occur that a record is omitted, or a record is mistakenly fetched. In the example, `rs2` mistakenly additionally contains `r2`.

In step ②, we translate the optimized query to an unoptimized query of the form `SELECT (`ϕ `IS TRUE) FROM t0`. This query lacks a `WHERE` condition. Thus, the DBMS must fetch every record in the selected table, which effectively disables most of the optimizations that the DBMS could apply. Furthermore, this query evaluates ϕ as a boolean predicate on every record in the table. This predicate should evaluate to `TRUE` for every record that is contained in the result set of the optimized query (*i.e.*, for which the `WHERE` clause evaluates to `TRUE`), because the predicate must consistently yield the same value, regardless of where it is used. The result set thus must contain two records with `TRUE`, and one record with `FALSE`.

In step ③, we pass both queries to the DBMS and compare the two result sets (i.e., `rs1`/`rs2` with `rs3`). For the optimized query, we count the number of records, that is, |`rs1`| = 2 for the correct

Listing 2: Join clauses can be copied during translation.

```
① SELECT * FROM t0 LEFT JOIN t1 ON t0.c0 = t1.c0 JOIN t2
     ON t2.c0 > t0.c1 WHERE t2.c0 = 5;
② SELECT ((t2.c0 = 5) IS TRUE) FROM t0 LEFT JOIN t1 ON
    t0.c0 = t1.c0 JOIN t2 ON t2.c0 > t0.c1;
```

execution and $|\texttt{rs2}| = 3$ for the incorrect execution. For the unoptimized query, we count the number of TRUE values in the result set, that is, $|\sigma_{column_1=TRUE}(r2) = 2|$, which should correspond to the number of records that are fetched for the optimized query. Since $2 \neq 3$ for the incorrect execution, NoREC detects a bug in the query optimizer. Note that we consider only the cardinalities of the result set for the optimized query and for how many rows the expression evaluates to TRUE for the unoptimized query to validate the DBMS. Our empirical evidence demonstrates that this suffices to find all optimization bugs. For completeness, Section 3.3 describes how NoREC can be extended to also validate the records' contents.

3.2 Translating the Query

Translating an optimized query to an unoptimized one is a straightforward, automatic procedure. As illustrated in Figure 1 step ②, it requires moving the condition in a WHERE clause to after the select statement, so that it is executed on every row in the table. As detailed next, the basic approach can be extended to cover additional features of the DBMS.

Multiple tables. In a FROM clause, multiple tables can be specified from which records are fetched, which are typically joined by a predicate in the WHERE clause. Although the previous examples only referred to a single table, our approach directly applies to multiple tables without any modifications.

Join clauses. Besides WHERE clauses, also JOIN clauses can be used to join two tables. For example, consider query ① in Listing 2, which shows an example with one (inner) JOIN and one LEFT JOIN. The ON clause for inner JOINs specifies that only those records should be fetched for which the condition evaluates to TRUE for records in both tables (*i.e.*, as if the predicate would have been specified in a WHERE clause). A LEFT JOIN fetches all records that an inner JOIN fetches; in addition, it fetches all records from the left table that do not have a matching record in the right table, by assuming selected columns from the right table to be NULL. These, and the other types of joins (*e.g.*, NATURAL JOINs, RIGHT JOINs, and FULL JOINs) can be left unmodified during translation. Query ② shows that only the WHERE condition `t2.c0 = 5` was moved after the SELECT clause, and that the JOINs were copied. An alternative strategy that could find additional bugs in joins would be to move their ON clauses as well, which would require translating them to multiple unoptimized queries (see Section 5).

ORDER BY. ORDER BY clauses do not influence the cardinality of the result set. Thus, the unoptimized query can either omit or replace it during translation, to test for bugs related to this feature.

GROUP BY. GROUP BY clauses group records with same values and are often used in combination with aggregate and window functions. These clauses, if present in the optimized query, can be copied to the unoptimized query. If so, an additional query is required to sum up the intermediate counts of the individual groups, assuming that an aggregate function is used to sum up the records for which the expression evaluates to TRUE (*cf.* Section 3.3).

Listing 3: We alternate between two strategies for determining the count for the optimized query.

```
① SELECT * FROM t0 WHERE ϕ -- while (rs.next()) count++
② SELECT COUNT(*) FROM t0 WHERE ϕ  -- count=rs.getInt(1)
```

Listing 4: We use an aggregate function to determine the count for the unoptimized query.

```
SELECT SUM(count) FROM (SELECT ϕ IS TRUE) as count FROM
    t0 -- count = rs.getInt(1)
```

3.3 Determining the Row Count

Figure 1, step ③ does not illustrate how to compute the counts for the optimized and unoptimized queries. We apply different strategies for this. The naive approach is to iterate through the result set to determine the count, which is applicable for both queries. The second, more efficient strategy—the performance gain varies on various parameters, such as the number of rows in the database—relies on aggregate functions provided by the DBMS to retrieve the count, but might result in bugs being overlooked, since the increased complexity of the query might make optimizations inapplicable. To balance performance and bug-finding capabilities, we alternate between both strategies.

Optimized query. Listing 3 demonstrates the two ways how we compute the count for the optimized query from Figure 1. Query ① represents the naive approach. For this query, the DBMS returns a result set rs, through which SQLancer iterates to determine the count. Query ② uses COUNT(*) to count the number of records by relying on the DBMS for this. This is more efficient because the DBMS might optimize the query, and also because the overhead for crossing the boundaries between the DBMS and SQLancer is avoided [5]. SQLancer only needs to retrieve the count from the single record in the result set rs returned by the DBMS.

Unoptimized query. For the unoptimized query, we assume that since the DBMS is unable to optimize the query, it is unable to optimize an aggregate function applied to it as well. Since using an aggregate function is more efficient, we use only this strategy (see Listing 4). The SUM() function adds up the predicate values by interpreting TRUE as one, and FALSE as well as NULL as zero. DBMS such as PostgreSQL and CockroachDB do not provide implicit conversions from booleans to integers, and require an additional cast.

Records content. Our basic idea can be extended to check the records' contents. To this end, the query generated by step ② must list each column in addition to the predicate. The records for which the predicate evaluates to TRUE for the unoptimized query can then be compared with those fetched for the optimized query in step ③. However, retrieving and comparing the result sets makes it necessary to use the slower naive strategy presented above. Checking the records' contents allowed us to find an additional bug in an SQLite extension, albeit not in its optimizer. We speculate that doing so was not more effective because we are unaware of any optimizations that transform the fetched content. Furthermore, while it is

possible that a DBMS returns an incorrect result set with the correct cardinality, our empirical evidence suggests that such bugs are unlikely to occur.

3.4 Corner Cases and Limitations

We tested a large subset of each DBMS' functionality and, in the process, identified general limitations as well as three SQLite corner cases that need to be specially treated by our approach. We do not consider these limitations to be essential, as they did not hinder the approach in finding bugs.

Ambiguous queries. SQL queries can be ambiguous, and thus it might be possible that a DBMS returns a different result for the optimized query than the unoptimized one, which was also a challenge for previous work [30]. In practice, we found subqueries to be problematic, especially when comparing the result of a subquery that might return more than one record with a value. Thus, we decided to disable the generation of subqueries and will consider generating unambiguous subqueries as part of future work.

Nondeterministic functions. A query might be unambiguous, but yield a different result between the optimized and unoptimized queries due to nondeterministic functions. Such functions include random number generators and those that return the current time. To prevent false positives, we disabled their generation.

Short circuit evaluation. Our approach is not applicable to detect bugs where an optimization results in an exception or error being "optimized way". This is due to SQL not specifying whether the `AND` and `OR` operators must short-circuit. We found that DBMS can handle this inconsistently between the optimized and unoptimized query. Consider a predicate ϕ_{ok} `AND` ϕ_{err}, where ϕ_{err} results in an error when executed. If ϕ_{ok} is executed first and yields `FALSE`, the DBMS might avoid also evaluating ϕ_{err}, causing the statement to execute without errors. Otherwise, the execution of ϕ_{err} results in an error. Consequently, our approach cannot detect incorrect optimizations that prevent expected errors to occur.

Other features. Our approach does neither directly apply to `DISTINCT` clauses nor to queries that compute results over multiple records such as aggregate as well as window functions, which is also a limitation that affects PQS. Also these features are optimized, meaning that their implementation might be affected by optimization bugs as well. We believe that our high level idea of translating an optimized to an unoptimized query could also be extended to be applicable in this context.

Number comparisons in SQLite. One of the three SQLite corner cases that caused problems was that SQLite3 considered floating-point numbers and integers that represent the same value to be equal, also when using the `DISTINCT` keyword, which caused inconsistent results. In Listing 5, the `DISTINCT` keyword in the view `v0` resulted in only one of the records being fetched—which one is unspecified and differed between the optimized and unoptimized query. For query ①, `0` was fetched; thus, the string concatenation yielded the value `00.1`, which evaluated to `TRUE`. For query ②, `0.0` was fetched from the view, which resulted in the concatenated string `0.00.1`, which evaluated to `FALSE`. Since such false positives were uncommon in SQLite3, and not present in the other DBMS, we initially manually filtered out such false positives, but then introduced an option to avoid generating `DISTINCT` keywords in views.

Listing 5: The `DISTINCT` keyword in views can cause inconsistent results in SQLite.

```
CREATE TABLE t0(c0);
INSERT INTO t0(c0) VALUES (0.0), (0);
CREATE VIEW v0(c0) AS SELECT DISTINCT c0 FROM t0;
① SELECT COUNT(*) FROM v0 WHERE v0.c0  0.1; -- 1
② SELECT (v0.c0  0.1) IS TRUE FROM v0; -- 0
```

Input columns in SQLite. The second SQLite corner case concerned the `dbstat` extension in SQLite (see Listing 6). The `WHERE` clause `stat.aggregate = 1` set an configuration option to `TRUE`, which changed the behavior of the query and causes a record to be fetched. When we used this predicate directly after the `SELECT` clause, however, the column was not used as an input and no record was fetched. We addressed this by avoiding the generation of clauses that set the configuration option for this specific column and extension.

Listing 6: Input columns in SQLite can change the behavior of queries

```
CREATE VIRTUAL TABLE stat USING dbstat;
SELECT * FROM stat WHERE stat.aggregate = 1; -- fetches
    one record
SELECT stat.aggregate = 1 FROM stat; -- FALSE
```

Ambiguous `GROUP BY`s in SQLite. The third SQLite corner case was given by ambiguous `GROUP BY`s in a view, which caused problems in combination with other features (such as optimizer hints, see Listing 7). All other DBMS that we tested prohibited such ambiguous `GROUP BY`s and returned an error for the view creation. In our complete testing period, we encountered such cases seldom, which is why we did not address this in SQLancer.

Listing 7: Ambiguous `GROUP BY`s in SQLite can cause inconsistent results

```
CREATE TABLE t0(c0, c1, c2, PRIMARY KEY(c2)) WITHOUT
    ROWID;
CREATE INDEX i0 ON t0(CAST(c1 AS INT));
CREATE VIEW v0 AS SELECT 0, c0 FROM t0 GROUP BY 1 HAVING
    c2;
INSERT INTO t0(c2) VALUES('');
INSERT INTO t0(c1, c2) VALUES(1, 1);
SELECT * FROM v0 WHERE UNLIKELY(1); -- {}
SELECT UNLIKELY(1) FROM v0; -- TRUE
```

3.5 Query and Database Generation

The random and targeted generation of databases [4, 7, 23, 26, 32, 38] and queries [2, 9, 30, 37, 42, 49, 58] for different workloads and purposes has been widely explored, and is not a contribution of this paper. Our approach can be applied based on any randomly-generated or existing database. It can also be applied to any random query generator that prevents the generation, or ignores errors in the corner cases described in Section 3.4. Thus, we explain our database and query generator only for completeness.

In our work, we base the generation of databases and queries on SQLancer [46], which we extended to cover additional DBMS (*i.e.* CockroachDB and MariaDB), as well as SQL features (*e.g.* additional data types, operators, and functions). SQLancer generates a database by randomly creating tables, indexes, inserting data, as well as by updating and deleting data to stress the DBMS in an attempt to increase the chances of finding bugs. The core part of SQLancer's random query generation is the generation of random expressions, which we use in `WHERE` and `JOIN` clauses. SQLancer generates these expressions heuristically, by selecting one of the applicable options. The applicable options are specific to a given DBMS, since DBMS vary in which operators they support and which implicit conversions they perform. The generation of the expressions is based on the grammar of the respective DBMS and the schema of the current database (to generate valid references to columns and tables).

4 EVALUATION

The goal of our evaluation was to demonstrate the effectiveness and generality of our approach. To this end, we tested NoREC on four widely-used DBMS: SQLite, MariaDB, PostgreSQL, and CockroachDB. As part of this, we extended SQLancer by a database and query generator for MariaDB as well as CockroachDB, and enhanced these components for SQLite and PostgreSQL (see Section 3.5). To maximize our real-world impact, we invested significant time and effort over a five-month period, which allowed us to find 159 true, previously unknown bugs. Furthermore, we analyzed the bug reports in order to obtain a better understanding on which kinds of bugs NoREC can find. Since PQS is the closest-related work, we compared PQS with NoREC.

4.1 Methodology

Tested DBMS. We focused on testing four important, widely-used DBMS: SQLite, PostgreSQL, MariaDB, and CockroachDB (see Table 1). According to the DB-Engines Ranking [17], the Stack Overflow's annual Developer Survey [40], and GitHub, these DBMS are among the most popular and widely-used DBMS. SQLite is the most widely deployed DBMS overall, used in most major web browsers, mobile phones, and embedded systems. The authors of SQLite speculate that over one trillion SQLite databases are in active use [53]. MySQL ranks on top of most popularity rankings. However, MySQL's binaries and its source code is provided only for release versions, which are typically published every 2-3 months, which makes it tedious to filter out test cases that trigger the same underlying bug, as also noted previously [46]. Furthermore, only some of the bugs found by PQS have been fixed, providing fewer incentives to test this DBMS. Thus, we decided to test MariaDB, which is a fork of MySQL, and uses an open-source development process. Since MariaDB shares much code with MySQL, we believe that the results are similar to those that would be obtained when testing MySQL. PostgreSQL is also a popular DBMS; it seems to be more robust than most other DBMS, and the PQS work could find only a single logic bug in it [46]. CockroachDB [56] is a recent commercial NewSQL DBMS [41]. It has received much attention and is highly popular on GitHub, although it has a low rank on the other popularity rankings. We tested only CockroachDB's free community edition, and not the commercial enterprise edition.

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

DBMS	Popularity Rank			LOC	First Release
	DB-Engines	Stack Overflow	GitHub Stars		
SQLite	11	4	1.5k	0.3M	2000
MariaDB	13	7	3.2k	3.6M	2009
PostgreSQL	4	2	6.3k	1.4M	1996
CockroachDB	75	-	17.7k	1.1M	2015

Testing focus. The developer's reaction times was a significant factor that determined on which DBMS we concentrated our testing efforts. The SQLite developers were most responsive in fixing bugs; typically, they would fix a bug within hours of us reporting it. Thus, we invested significantly more time into testing SQLite than for the other DBMS. Besides testing SQLite's core, we tested three important extensions that are included as part of SQLite's source code, but need to be enabled during build time. One extension enables Full Text Search (FTS) for SQLite, which was subject to extensive investigations by security researchers [57], as it is, for example, enabled in Google Chrome. R-Tree is an important index structure for spatial objects that is designed for efficiently supporting range queries [25]. `DBSTAT` is a *virtual table* that allows querying information about the content of a SQLite database. We also invested significant effort into testing PostgreSQL; however, we were unable to find any interesting bugs, which is why the developer's reaction time was insignificant. The CockroachDB developers quickly confirmed our bugs, and fixed many of them within days, especially those in the query optimizer. The MariaDB developers quickly confirmed our bugs; however, only one was fixed. We stopped testing MariaDB after reporting the initial bugs, due to the difficulty of filtering out duplicates.

Existing testing efforts. All DBMS are extensively tested, which we want to illustrate based on SQLite, and CockroachDB, which both documented their testing efforts. SQLite likely has the most impressive testing effort, which is documented on the SQLite homepage [52]. The SQLite developers follow a design process inspired by the DO-178B guidelines [59], which are concerned with the safety of safety-critical software used in certain airborne systems. They achieve 100% modified condition/decision coverage [59], which implies that every branch has been taken and falls through at least once. Besides, they employ out-of-memory testing, I/O error testing, crash testing, compound failure testing, fuzz testing, and dynamic analysis [52]. We believe that CockroachDB is an interesting target, because the developers have put significant effort into developing and using automatic testing techniques, which they run as part of their continuous integration. For example, they have been running a grammar-based fuzzer on CockroachDB that has found 70 bugs such as crashes and hangs within 3 years [28, 29]. In addition, they ported SQLSmith to Go to use it as an alternative query generator, which found over 40 bugs [29]. However, as noted by them [29], "[t]his kind of fuzz testing is not able to deduce correctness", which is the gap that we are filling with NoREC. When we reported bugs, they actively worked on enhancing their testing infrastructure

Listing 8: A bug in the IN affected expressions with affinity.

```
CREATE TABLE t0(c0 INT UNIQUE);
INSERT INTO t0(c0) VALUES (1);
SELECT * FROM t0 WHERE '1' IN (t0.c0); -- {1} 🐞 {} ✔
```

to find similar correctness bugs based on domain knowledge and random testing techniques.

Testing methodology. We implemented our approach iteratively and applied it to the DBMS under test after each iteration. Typically, we added a new feature to the random query generator (*e.g.*, a new operator) or database generator (*e.g.*, a new data type), after which we continued testing the DBMS. While some bugs were found seconds after implementing a feature, others were found only after weeks. After finding a bug, we reduced the test case. Although special query reducers have been proposed [30, 50], we found that C-Reduce [43], a tool that was originally developed for reducing C/C++ programs, was sufficient for our use case. We also manually reduced and canonicalized test cases, to reduce the developer's debugging effort. After excluding potential duplicates, we reported issues on the bug tracker, mailing list, or via a private report (when we considered the bug to potentially be a security issue). We did not analyze any potential security issues, since the focus of this work are optimization bugs. Until a bug was fixed, we tried to avoid generating patterns that triggered the bug. We invested significant time and effort in testing, as well as in triaging and reporting bugs, and opened a total of 168 bug reports. Due to the iterative implementation and deployment of our tool, we cannot provide any detailed statistics on the total run time or efficiency.

4.2 Selected Bugs

Next, we show a selection of bugs found by NoREC to give an intuition on what kind of *interesting* bugs it can find. These bugs are necessarily biased—we found many less interesting bugs, but also other interesting ones that we had to omit. The full list of bugs can be found as part of the supplementary material supplied with the manuscript. For brevity, we omit the unoptimized query where it can be directly derived from the optimized one. Rather, we highlight the actual, incorrect result with a 🐞 symbol, and the expected result with a ✔ symbol.

4.2.1 *Selected SQLite Bugs.*

Incorrect `IN` *optimization.* The SQL `IN` operator allows checking whether a value on the left side is contained in a set of values on the right side. Previously, SQLite implemented an optimization that transformed an expression of the form `X IN (Y)` to `X=Y` (note that `Y` is a single value). For the `=` operator, SQLite performs implicit type conversions based on the *affinity* of an operand (*e.g.* the column type), while it is not supposed to perform them for the `IN` operator. We thus found that this optimization is incorrect in the presence of affinity conversions (see Listing 8), as it caused SQLite to mistakenly convert the string `'1'` to an integer in the query, thus unexpectedly fetching a record. We found other, similar bugs related to affinity conversions (*e.g.*, a bug in the constant propagation). We believe that affinity conversions are difficult to reason about, and our findings seem to demonstrate that this mechanism is error prone.

Listing 9: Commuting the operator resulted in the partial index being mistakenly used.

```
CREATE TABLE t0(c0 COLLATE NOCASE, c1);
CREATE INDEX i0 ON t0(0) WHERE c0 >= c1;
INSERT INTO t0 VALUES('a', 'B');
SELECT * FROM t0 WHERE t0.c1 <= t0.c0; -- {}🐞, {a|B}✔
```

Listing 10: A bug in the vectorization engine caused a record to be omitted.

```
SET SESSION VECTORIZE=experimental_on;
CREATE TABLE t1(c0 INT);
CREATE TABLE t0(c0 INT UNIQUE);
INSERT INTO t1(c0) VALUES (0);
INSERT INTO t0(c0) VALUES (NULL), (NULL);
SELECT * FROM t0, t1 WHERE t0.c0 IS NULL; -- {NULL|0}🐞 {
    NULL|0, NULL|0}✔
```

Listing 11: A bug in the handling of filters unexpectedly caused a record to be fetched.

```
CREATE TABLE t0(c0 BOOL UNIQUE, c1 BOOL CHECK (true));
INSERT INTO t0(c0) VALUES (true);
SELECT * FROM t0 WHERE t0.c0 AND (false NOT BETWEEN
    SYMMETRIC t0.c0 AND NULL AND true); -- {TRUE}🐞 {}✔
```

Operator commuting disregards `COLLATE`. We found a bug where commuting an operator mistakenly resulted in matching an inapplicable partial index (see Listing 9). The `COLLATE NOCASE` clause specifies that when this column is used in string comparisons, the casing of strings should be disregarded; however, since the `c1` column is used on the left hand side and lacks a `COLLATE`, the casing is assumed to be relevant. Thus, the lowercase characters `'a'` is assumed to be greater than the uppercase `'B'`, making the predicate yield `TRUE`. However, the record was not fetched. The cause for the bug was that SQLite commuted the operator, while updating the expressions' `COLLATE`s, which subsequently caused it to match the partial index, since insufficient information was preserved to verify whether the expression correctly qualifies the index. The bug was fixed by adding logic to maintain this information.

4.2.2 *Selected CockroachDB Bugs.*

Vectorization engine bug. We found 11 bugs related to CockroachDB's vectorization engine, one of which is illustrated by Listing 10. The query is expected to fetch two records, because the two tables are joined without filtering out records, effectively computing the table's cross product (`|t0|*|t1| = 2*1 = 2`). However, only one record was fetched. The core reason for this bug was that the empty set of equality columns for the join between the two tables was handled incorrectly for hash joins by the vectorized execution engine.

Incorrect handling of filters. We found a bug that exposed that CockroachDB, in rare cases, incorrectly handled `CHECK` constraints, which are used to refine the ranges in filters, causing the query in Listing 11 to incorrectly fetch a record, even though the predicate should evaluate to `NULL`. While this specific bug was fixed by introducing a missing normalization rule, the underlying cause for the bug was fixed in a subsequent commit that extended and refactored CockroachDB's index constraints library.

Listing 12: A bug in the range scan resulted in a row being omitted.

```
CREATE TABLE t0(c0 INT UNIQUE);
INSERT INTO t0 VALUES(NULL),(NULL),(NULL),(NULL),(1),(0);
SELECT * FROM t0 WHERE c0 < '\n2'; -- {0} {0, 1}
```

Listing 13: A predicate comparing a float-point number with an integer unexpectedly evaluated to TRUE.

```
CREATE TABLE t0(c0 INT);
INSERT INTO t0 VALUES (1);
CREATE INDEX i0 ON t0(c0);
SELECT * FROM t0 WHERE 0.5 = c0; -- {1} {}
```

4.2.3 *Selected MariaDB Bugs.*

Incorrect string range scan. We found a bug in MariaDB where a range scan on an index was applied incorrectly for a string comparison (see Listing 12). As explained by the MariaDB developers, the optimizer incorrectly constructed the range `NULL < x <= 0` for the range scan, even though the upper limit should be 2, leading to only one row being fetched. The reason for that was that space characters like `\n` were handled inconsistently. This bug also affected MySQL, and was the only bug that we reported that was fixed by the MariaDB developers.

Incorrect number comparison. We found a bug where a comparison of a floating-point number with an integer column, on which an index is created, yielded an incorrect result (see Listing 13). The comparison `0.5=c0` should evaluate to `FALSE`, because `c0` should, after an implicit type conversion, evaluate to the value `1.0`, and `0.5≠1.0`. However, the query unexpectedly fetched the single record stored in the table. While this bug was quickly confirmed and reproduced for both MariaDB and MySQL, it has not been addressed yet.

4.3 Bugs Overview

General bug statistics. Table 2 shows the number of bugs that we reported and their status. Out of the 168 bug reports, 159 turned out to be previously-unknown *true bugs*. 141 of these bugs were addressed by code changes, demonstrating that the developers took our bug reports seriously. 14 bugs were verified but have not been addressed yet, and 3 bug reports were addressed by documentation changes. 9 turned out to be *false bugs*. Out of these, 7 were not considered to be bugs, either because an internal error that we considered to be unexpected was actually expected by the developers, or because we were not yet aware of the limitations of the approach (see Section 3.4). As we tested the latest version of each DBMS, only 2 bugs turned out to be already known.

Oracles. Table 3 shows the oracles that we used to find the bugs. We found 51 bugs with the NoREC oracle, which was the primary focus of our work. Besides, we identified 58 bugs through unexpected internal errors, either while creating the database, or when sending queries to it. We found these bugs by annotating a list of expected errors to each SQL statement [46]. SQLancer identified also many crash bugs, since it implicitly acts as a grammar-based fuzzer. We built the debug versions of SQLite and PostgreSQL, which allowed us to find 27 debug assertion failures. Furthermore, we found 23 crash bugs—which also included hang bugs—in release builds (but not necessarily in released versions of the DBMS).

Table 2: We found 159 bugs in SQLite, MariaDB, PostgreSQL, and CockroachDB, 141 of which have been fixed.

			Closed	
DBMS	Fixed	Verified	Intended	Duplicate
SQLite	110	0	6	0
MariaDB	1	5	0	1
PostgreSQL	5	2	1	0
CockroachDB	28	7	0	1

Table 3: We found 51 bugs with the NoREC oracle, 58 through unexpected errors, 27 by debug assertion failures, and 23 by crashes that occur in release version.

			Crash	
DBMS	Logic	Error	Release	Debug
SQLite	39	30	15	26
MariaDB	5	0	1	0
PostgreSQL	0	4	3	1
CockroachDB	7	24	4	0

Additional clauses. Section 3.2 mentions that optionally, `ORDER BY` and `GROUP BY` clauses can be generated, to further stress the query optimizer. We found one logic bug, and one crash bug with an `ORDER BY` clause. We found only one error bug using a `GROUP BY` clause. Overall, these two clauses did not contribute much to NoREC's bug-finding capabilities; however, their implementation requires little effort, which might still justify their implementation.

SQLite. We found most bugs in SQLite, which is expected, since we invested most effort and energy into testing it. Out of the 110 SQLite bugs, 71 affected the SQLite core. A smaller portion affected extensions; we found 13 bugs in RTREE, 24 in FTS, and 2 in DBSTAT. Note that some bugs that we found in these extensions affected virtual tables in general, on which these and other extensions are based. While we were testing SQLite, the developers added support for *generated columns* [51], which are columns that are computed based on other columns. After this feature was merged to trunk, but before it was released, we found 22 bugs in it, contributing significantly to its correctness. Besides logic bugs, 26 bugs manifested themselves as debug assertion failures. This high number can be explained by previous work such as PQS having omitted testing them. A number of these assertions did not indicate real bugs in the SQLite core; rather, they indicated the omission of corner cases in the testing logic that SQLite uses [55]. The high number of crashes in release builds is surprising, considering that PQS found only 2 crash bugs in SQLite [46]. One main reason is the aforementioned generated column feature, in which we found 9 crash bugs. We also found one hang bug in FTS, one bug that involved a trigger, two bugs in RTREE, and two bugs in window functions.

PostgreSQL. Although we invested significant effort into testing PostgreSQL, we found only 8 bugs in it. None of these bugs was an optimization bug. This is consistent with previous findings; for example, PQS could find only a single logic bug in this DBMS [46]. We believe that one significant reason for that is that PostgreSQL is very restrictive in what input it accepts compared to the other DBMS. Richard Hipp, the main SQLite developer, also noted that PostgreSQL in particular is a high-quality DBMS, which has had few bugs and noted that one possible reason could be their very elaborate peer review process [59].

CockroachDB. We found 35 bugs in CockroachDB. In 15 cases, our bug reports relied on experimental features. Of these, 11 bugs affected the vectorizer engine (see Listing 10). Out of the 24 error bugs, 17 were due to internal errors, for which execution resulted in displaying a stack trace along with information on where to report the bug, while the server stayed responsive. Based on our bug reports, the CockroachDB developers actively added testing infrastructure and reviewed code to detect similar bugs, demonstrating that they took our bug reports seriously. For example, one of the duplicate bug report was due to an open issue that acknowledged that the issue was found based on one of our bug reports.

MariaDB. We found 6 bugs in MariaDB. All bugs were quickly confirmed by the MariaDB developers, and three of the bugs were reproduced also for MySQL. However, only one bug was fixed within a duration of three months, which is why we stopped testing MariaDB. Since we invested only little time in testing MariaDB, we believe that our approach could find additional bugs in it.

4.4 Comparison to PQS

PQS is the state of the art in finding logic bugs in DBMS, which is why we want to compare NoREC's effectiveness with it. We expected PQS to find a broader class of bugs, because our aim for NoREC was to find optimization bugs. To the best of our knowledge, there is no other publicly available tool that could detect logic bugs in DBMS to which we could compare.

Evaluation challenges. Fairly comparing the effectiveness between PQS and NoREC is difficult due to various reasons. First, the implementation effort of PQS is significant, and limited the authors to a core subset of the respective DBMS's supported SQL dialect [46]. For example, in PQS, a single comparison operator alone covers already more than 200 LOC, since it needs to support comparing arbitrary data types, which also involves implicit conversions for DBMS like SQLite or MySQL. In contrast, the implementation of the NoREC oracle consists of less than 200 LOC, and also allows finding bugs in the optimization of complex operators and functions. Nevertheless, we believe that a fair evaluation should disregard the amount of time that was invested into implementing the respective approach. Another challenge is that a different set of DBMS were tested. While both PQS and NoREC were evaluated on SQLite and PostgreSQL, PQS was also evaluated on MySQL, while we evaluated NoREC on MariaDB, and CockroachDB. Overall, this inhibited us from doing an automatic comparison.

Methodology. To provide a fair comparison, we performed a manual quantitative and qualitative comparison. We noticed that a test case for NoREC can typically be converted directly to an equivalent PQS test case that triggers the bug, if it can be reproduced by PQS. In fact, we can take the unmodified query with the predicate in the `WHERE` clause and check if the bug can be reproduced by selecting an applicable pivot row. Likewise, a PQS test case can often be converted to an equivalent NoREC test case, by performing the translation step to an unoptimized query. Based on this observation, we manually tried to create equivalent test cases where possible, which we then evaluated on the version of the DBMS in which the bug was found. While we cannot completely rule out misclassifications that might be due to overlooking that a bug could be reproduced by another query, we believe that the majority of cases were clear. Note that we considered only bugs found by the NoREC oracle—we expect that the errors and crashes can be triggered with either of the approaches. Overall, we investigated (1) the 51 bugs found by the NoREC oracle to check if they could have been found by PQS as well and (2) the 61 bugs found by PQS to check whether they could have been found by NoREC.

Bugs found only by NoREC. PQS could detect 56.9% of the bugs that were found by NoREC. Specifically, NoREC detected 4 bugs for which the result set unexpectedly contained or missed duplicate records. PQS conceptually cannot detect such bugs, because it validates only whether a randomly-selected record, which is indistinguishable from other duplicate rows, is part of the result set. NoREC triggered 5 bugs by aggregate functions that are used for counting the rows (*i.e.*, `SUM` and `COUNT`). PQS' main oracle relies on only checking a single row at a time, which conceptually hinders PQS in detecting bugs related to these aggregate functions. We found 13 cases where records were mistakenly fetched. PQS did not detect this class of bugs, because it only checks for bugs where the pivot row is mistakenly omitted from the result set. We believe that this not a fundamental limitation of PQS, because it could be extended to also generate queries that guarantee to omit the pivot row, enabling PQS to also detect such bugs. Taking this into account, PQS could detect 82.4% of the bugs that NoREC found. A caveat that was already mentioned is that some other bugs can be detected only by a close-to-complete implementation of PQS; as noted, we did not consider this limitation in this analysis, since it is arguable which cases would be deemed to involve too much of an implementation effort.

Bugs found only by PQS. In total, NoREC could have found 52.7% of the bugs that PQS found. By far the most common kind of bugs that were detected by PQS, but not by NoREC, stem from the incorrect implementation of operators, functions, and other features (especially affinity conversions in SQLite), both in the optimized and unoptimized case. NoREC failed to detect 18 such bugs. 3 bugs could not be triggered by NoREC, since they relied on a `DISTINCT` query, which was disregarded by NoREC, but could be supported by translating it to a `GROUP BY` clause in the unoptimized case. While PQS, like NoREC, is in general unable to detect bugs in aggregate functions, since it operates on a single pivot row, it can do so when a table contains only a single row, which allowed it to detect 3 bugs. PQS could also find 1 bug that was exposed when using `INTERSECT`, which PQS uses to efficiently check containment, similarly to how NoREC uses the aggregate functions for counting. 1 bug was triggered based on a predicate in an `ON` clause of a `LEFT`

JOIN. NoREC failed to discover that bug, because it copies left joins and predicates when translating the query. As outlined in Section 5, we believe that our approach might be enhanced by implementing additional translation schemes for joins.

Discussion. The comparison has demonstrated that PQS can find a number of bugs that NoREC cannot find. Although NoREC could find classes of bugs that PQS is unable to find as well, such as duplicate records, it is mostly restricted to finding optimization bugs. However, PQS incurs significant implementation overhead, since every operator and function to be tested needs to be implemented, for every DBMS that should be tested. In contrast, NoREC relies on a straightforward translation process, and is applicable to any database and query generator that can address the limitations mentioned in Section 3.4. Due to the low implementation effort, NoREC has successfully found a wide range of optimization bugs, even for SQLite, which has recently been comprehensively tested by PQS. Thus, we envision that DBMS could be tested by a combination of both approaches. PQS could be used to test many basic operators and functions, and, being an exact oracle, it can help in establishing a ground truth. Then, NoREC could be used to find the lingering optimization bugs in parts not comprehensively tested by PQS.

5 DISCUSSION

Reception by the DBMS developers. Developers of multiple DBMS told us that they appreciated our bug reports. The SQLite homepage even highlights our efforts at https://www.sqlite.org/testing.html: "*One fuzzing researcher of particular note is Manuel Rigger, [...]. Most fuzzers only look for assertion faults, crashes, undefined behavior (UB), or other easily detected anomalies. Dr. Rigger's fuzzers, on the other hand, are able to find cases where SQLite computes an incorrect answer. Rigger has found many such cases. Most of these finds are fairly obscure corner cases involving type conversions and affinity transformations, and a good number of the finds are against unreleased features. Nevertheless, his finds are still important as they are real bugs, and the SQLite developers are grateful to be able to identify and fix the underlying problems. Rigger's work is currently unpublished. When it is released, it could be as influential as Zalewski's invention of AFL and profile-guided fuzzing.*"

Bug importance. We believe that we found many critical bugs that are likely to affect real users. However, we also acknowledge that many of the other bugs that we found can be triggered only by an unlikely combination of operators or features. Even those seemingly unimportant bugs can affect users due to the widespread use of these DBMS, for which we found evidence on the SQLite mailing list, where a user reported an incorrect result for a query with a complex WHERE predicate. When confronted by another SQLite user, the reporter of the bug defended themselves with the following [13]: "*I might not spell it like that myself, but a code generator would do it (and much worse!). This example was simplified from a query generated by a Django ORM queryset using* `.exclude(nullable_joined_table__column=1)`*, for instance.*" The bug had already been fixed in the latest development version, because we previously reported the same underlying bug [44]. We found it with a query on a view and a predicate NOTNULL NOTNULL, which would unlikely to be written by a programmer as well. We speculate that also other users might be affected by such seemingly-obscure bugs; however, finding the root cause in such cases is difficult, especially when the query is generated by middleware.

Handling of joins. Our translation process leaves JOIN clauses unmodified (see Section 3.2). Although this allowed us to find some bugs in their handling, translating them to a condition that is placed after the SELECT as well could uncover additional bugs. For inner joins, this is straightforward, because an ON predicate ϕ_1 and a WHERE predicate ϕ_2 can be translated to a predicate ϕ_1 AND ϕ_2. For other joins, this would be more involved. For example, for LEFT JOINs, a simple SELECT statement would no longer sufficient, but would require combining the results of multiple queries. We consider additional strategies in translating JOIN clauses as potentially useful and part of future work.

Code coverage and performance. At first sight, it might be interesting to evaluate the code coverage and run-time performance that NoREC achieves. However, neither helps to explain the approach's effectiveness. With respect to coverage, it was found that code coverage is not particularly useful for fuzzing DBMS, since high coverage for the core components (*e.g.*, the query optimizer) can be achieved quickly [30]. Furthermore, although the SQLite developers achieve 100% MC/DC coverage in their test suite, we could still find bugs in SQLite. The run-time overhead of the query generation and translation is negligible, which is why we did not measure it in detail. The run time is dominated by the DBMS, which needs to process the generated queries, as well as by the communication with the DBMS. The communication overhead was lower for SQLite than for the other DBMS, because SQLite is an *embedded* DBMS, that is, it runs within the application process of SQLancer.

Kind of bugs. 51 of the 159 true bugs that we found were optimization bugs. We believe that such bugs are most severe, since they are likely to go unnoticed by developers. Bugs caused by crashes, assertion failures, and error represent a large fraction of the overall bugs. However, they can be found by existing approaches, such as grammar-based fuzzers. Furthermore, such bugs typically cause the DBMS to terminate, thus signalling the user that the DBMS malfunctions for the given query. A possible conclusion might be that our testing approach detects also errors typically found by fuzzers, and that such bugs might be more common or easier to find than logic bugs (given that we do not have the ground truth on the total number of bugs).

Fully automatic approach. We claim that NoREC is fully automatic, as it finds bugs by repeatedly generating test cases and validating their result set, without requiring any human interaction. As discussed in Section 3.4, corner cases exist that have to be treated specially to ensure that only true bugs are reported. When NoREC detects a bug, it is helpful to reduce the generated test case so that it is minimal. For our evaluation, we first automatically reduced the test case, and then manually tried to reduce it further; we consider manual reduction optional, but convenient for the DBMS developers for debugging. Before reporting bugs, we manually checked the bug tracker to lower the chances of reporting duplicate bugs. In practice, it would be useful to suspend running NoREC after it reports a bug, and continue its execution after the bug is fixed.

6 RELATED WORK

Differential testing of DBMS. Differential testing discovers bugs by executing a given input using multiple versions of a system to detect differing outputs, which indicate a bug in one of these systems. Similarly to metamorphic testing, differential testing has proven to be effective in many domains [6, 16, 31, 33, 36, 61]. For DBMS, it was first applied by Slutz in a tool *RAGS* to find bugs by executing a query on multiple different DBMS and comparing their result sets [50]. While the approach was highly effective, it can only be applied to a small subset of common SQL. Gu et al. used options and hints to force the generation of different query plans, to then rank the accuracy of the optimizer based on the estimated cost for each plan [24]. Jinho et al. used differential testing in a system called *APOLLO* to find performance regression bugs in DBMS, by executing a SQL query on an old and newer version of a DBMS [30]. They found 10 previously-unknown performance regression bugs in SQLite and PostgreSQL. While our work uses metamorphic testing to find optimization bugs, conceptually, it can be interpreted as realizing differential testing by comparing the results of an optimizing, and non-optimizing version of a DBMS.

Other correctness oracles for testing DBMS. Pivoted Query Synthesis (PQS) is the state of the art in testing DBMS for logic bugs and is the most closely-related work [46]. It is both an automatic testing approach as well as an oracle, and is based on the idea of testing the DBMS' correctness on a randomly-selected *pivot row*. PQS has found close to 100 bugs in widely-used DBMS, demonstrating that it is highly effective. However, due to its high implementation effort, it might be infeasible to test all of a DBMS' supported operators and functions. NoREC is mainly applicable to finding optimization bugs, a subcategory of logic bugs, towards which PQS is geared. Due to its low implementation effort, it can find bugs in components for which the implementation of PQS would be too costly. ADUSA [32] is a query-aware database generator and generates input data as well as the expected result for a query, thus also tackling the correctness oracle problem for DBMS. It translates the schema and query to an Alloy specification, which is subsequently solved. The approach could reproduce various known and injected bugs in MySQL, HSQLDB, and also find a new bug in Oracle Database. We believe that the high overhead that solver-based approaches incur might inhibit such approaches from finding more DBMS bugs.

Random and targeted queries. Many query generators have been proposed for purposes such as bug-finding and benchmarking. SQLsmith is a widely-used, open-source random query generator, which has found over 100 bugs in widely-used DBMS [49]. Bati et al. proposed an approach based on genetic algorithms to incorporate execution feedback for generating queries [2]. SQLFUZZ [30] also utilizes execution feedback and randomly generates queries using only features that are supported by all the DBMS systems they considered. Khalek et al. proposed generating both syntactically and semantically valid queries based on a solver-backed approach [1]. The problem of generating queries that satisfy cardinality constraints on their sub-expressions was shown to be computationally hard [9, 37]. Consequently, a number of heuristic and approximate approaches to generating targeted queries were proposed [9, 37, 42, 58]. All these random-query generators can be used to find bugs such as crashes and hangs in DBMS. When paired with the test oracle proposed in this paper, they could also be used to find logic bugs.

Random and targeted databases. Many approaches have been proposed to automatically generate databases. Given a query and a set of constraints, QAGen [4, 34] generates a database that matches the desired query results by combining traditional query processing and symbolic execution. Reverse Query Processing takes a query and a desired result set as an input to then generate a database that could have produced the result set [3]. As discussed above, ADUSA is a query-aware database generator [32]. Gray et al. discussed a set of techniques utilizing parallel algorithms to quickly generate billions-record databases [23]. DGL was proposed as a domain-specific language to generate input data following various distributions and inter-table correlations based on *iterators* that can be composed [7]. Neufield et al. proposed to generate test data for tables with constraints by deriving a generator formula that is then translated to operators to generate the test data [38]. An improved database generation might enable NoREC to find additional bugs.

Metamorphic testing. Our approach is based on *metamorphic testing* [11]. Metamorphic testing addresses the test oracle problem by, based on an input and output of a system, generating a new input for which the result is known. This technique has been applied successfully in various domains [12, 47]. Central in this approach is the metamorphic relation, which can be used to infer the expected result. In this work, we combine a translation and counting mechanism to establish a metamorphic relation specifically geared towards detecting optimization bugs. A limitation of metamorphic testing is that it cannot establish a ground truth. For NoREC, this implies that both the unoptimized and optimized query can yield an incorrect result (see Section 4.4). Similarly, the optimized query could compute the correct result, rather than the unoptimized one.

7 CONCLUSION

This paper has proposed a general, highly-effective approach for detecting bugs in DBMS, which we termed Non-optimizing Reference Engine Construction (NoREC). The core insight of NoREC is that a given *optimized* query can be translated to an *unoptimized* query, which enables the construction of a test oracle that can detect *optimization bugs* by comparing the result sets of the two queries. While we believe that this work provides a solid foundation for correctness testing of DBMS, the basic idea of NoREC could be extended by using additional or alternative translation strategies for queries, for example, by translating predicates of `JOIN` clauses. As another example, queries could also be transformed in other ways, while still being expected to generate the same results (*e.g.*, by switching the operands of commutative operators). Furthermore, the efficiency and effectiveness of NoREC could be enhanced by pairing it with better database and query generators.

ACKNOWLEDGMENTS

We want to thank the DBMS developers for verifying and addressing our bug reports as well as for their feedback to our work. Furthermore, we are grateful for the feedback received by the anonymous reviewers, Martin Kersten, as well as by the members of the AST Lab at ETH Zurich.

REFERENCES

[1] Shadi Abdul Khalek and Sarfraz Khurshid. 2010. Automated SQL Query Generation for Systematic Testing of Database Engines. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. ACM, New York, NY, USA, 329–332. https://doi.org/10.1145/1858996.1859063

[2] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. 2007. A Genetic Approach for Random Testing of Database Systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, 1243–1251.

[3] Carsten Binnig, Donald Kossmann, and Eric Lo. 2007. Reverse Query Processing. *Proceedings - International Conference on Data Engineering*, 506–515. https://doi.org/10.1109/ICDE.2007.367896

[4] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. 2007. QAGen: Generating Query-Aware Test Databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 341–352. https://doi.org/10.1145/1247480.1247520

[5] Carl Friedrich Bolz, Darya Kurilova, and Laurence Tratt. 2016. Making an Embedded DBMS JIT-friendly. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. 4:1–4:24. https://doi.org/10.4230/LIPIcs.ECOOP.2016.4

[6] Robert Brummayer and Armin Biere. 2009. Fuzzing and Delta-Debugging SMT Solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT '09)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/1670412.1670413

[7] Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible Database Generators. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*. VLDB Endowment, 1097–1107.

[8] Nicolas Bruno, Surajit Chaudhuri, and Ravi Ramamurthy. 2009. Power Hints for Query Optimization. In *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE '09)*. IEEE Computer Society, USA, 469–480. https://doi.org/10.1109/ICDE.2009.68

[9] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. 2006. Generating Queries with Cardinality Constraints for DBMS Testing. *IEEE Trans. on Knowl. and Data Eng.* 18, 12 (Dec. 2006), 1721–1725. https://doi.org/10.1109/TKDE.2006.190

[10] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A Structured English Query Language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control (SIGFIDET '74)*. ACM, New York, NY, USA, 249–264. https://doi.org/10.1145/800296.811515

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

[12] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1, Article Article 4 (Jan. 2018), 27 pages. https://doi.org/10.1145/3143561

[13] And Clover. 2019. Bug submission: left join filter on negated expression including NOTNULL. https://www.mail-archive.com/sqlite-users@mailinglists.sqlite.org/msg117434.html

[14] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387. https://doi.org/10.1145/362384.362685

[15] E. F. Codd. 1972. *Relational Completeness of Data Base Sublanguages*. IBM Corporation.

[16] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *Proceedings of the 4th International Conference on NASA Formal Methods (NFM'12)*. Springer-Verlag, Berlin, Heidelberg, 120–125. https://doi.org/10.1007/978-3-642-28891-3_12

[17] DB-Engines. 2019. DB-Engines Ranking (July 2019). https://db-engines.com/en/ranking

[18] Bailu Ding, Sudipto Das, Wentao Wu, Surajit Chaudhuri, and Vivek Narasayya. 2018. Plan Stitch: Harnessing the Best of Many Plans. *Proc. VLDB Endow.* 11, 10 (June 2018), 1123–1136. https://doi.org/10.14778/3231751.3231761

[19] Ramez Elmasri and Sham Navathe. 2017. *Fundamentals of database systems*. Vol. 7. Pearson.

[20] Leo Giakoumakis and César A Galindo-Legaria. 2008. Testing SQL Server's Query Optimizer: Challenges, Techniques and Experiences. *IEEE Data Eng. Bull.* 31, 1 (2008), 36–43.

[21] Torsten Grabs, Steve Herbert, and Xin (Shin) Zhang. 2008. Testing Challenges for Extending SQL Server's Query Processor: A Case Study. In *Proceedings of the 1st International Workshop on Testing Database Systems (DBTest '08)*. Association for Computing Machinery, New York, NY, USA, Article Article 2, 6 pages. https://doi.org/10.1145/1385269.1385272

[22] Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)* 25, 2 (1993), 73–169.

[23] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. *SIGMOD Rec.* 23, 2 (May 1994), 243–252. https://doi.org/10.1145/191843.191886

[24] Zhongxian Gu, Mohamed A. Soliman, and Florian M. Waas. 2012. Testing the Accuracy of Query Optimizers. In *Proceedings of the Fifth International Workshop on Testing Database Systems (DBTest '12)*. ACM, New York, NY, USA, Article 11, 6 pages. https://doi.org/10.1145/2304510.2304525

[25] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/602259.602266

[26] Kenneth Houkjær, Kristian Torp, and Rico Wind. 2006. Simple and Realistic Data Generation. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB '06)*. VLDB Endowment, 1243–1246.

[27] William E. Howden. 1978. Theoretical and Empirical Studies of Program Testing. In *Proceedings of the 3rd International Conference on Software Engineering (ICSE '78)*. IEEE Press, Piscataway, NJ, USA, 305–311.

[28] Matt Jibson. 2016. Testing Random, Valid SQL in CockroachDB. https://www.cockroachlabs.com/blog/testing-random-valid-sql-in-cockroachdb/

[29] Matt Jibson. 2019. SQLsmith: Randomized SQL Testing in CockroachDB. https://www.cockroachlabs.com/blog/sqlsmith-randomized-sql-testing/

[30] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *Proc. VLDB Endow.* 13, 1 (Sept. 2019), 57–70. https://doi.org/10.14778/3357377.3357382

[31] Timotej Kapus and Cristian Cadar. 2017. 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)*. IEEE Press, Piscataway, NJ, USA, 590–600.

[32] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. 2008. 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)*. IEEE Computer Society, Washington, DC, USA, 238–247. https://doi.org/10.1109/ASE.2008.34

[33] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

[34] Eric Lo, Carsten Binnig, Donald Kossmann, M. Tamer Özsu, and Wing-Kai Hon. 2010. A framework for testing DBMS features. *The VLDB Journal* 19, 2 (01 Apr 2010), 203–230. https://doi.org/10.1007/s00778-009-0157-y

[35] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

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

[37] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. 2008. Generating Targeted Queries for Database Testing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 499–510. https://doi.org/10.1145/1376616.1376668

[38] Andrea Neufeld, Guido Moerkotte, and Peter C. Lockemann. 1993. Generating Consistent Test Data: Restricting the Search Space by a Generator Formula. *The VLDB Journal* 2, 2 (April 1993), 173–214.

[39] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 677–692. https://doi.org/10.1145/3183713.3183733

[40] Stack Overflow. 2019. Developer Survey Results 2019. https://insights.stackoverflow.com/survey/2019

[41] Andrew Pavlo and Matthew Aslett. 2016. What's Really New with NewSQL? *SIGMOD Rec.* 45, 2 (Sept. 2016), 45–55. https://doi.org/10.1145/3003665.3003674

[42] Meikel Poess and John M. Stephens. 2004. Generating Thousand Benchmark Queries in Seconds. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30 (VLDB '04)*. VLDB Endowment, 1045–1053.

[43] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

[44] Manuel Rigger. 2019. LEFT JOIN in view malfunctions with NOTNULL. https://www.sqlite.org/src/tktview?name=c31034044b

[45] Manuel Rigger and Zhendong Su. 2020. ESEC/FSE 20 Artifact for "Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction". https://doi.org/10.5281/zenodo.3947858

[46] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis.

[47] Sergio Segura and Zhi Quan Zhou. 2018. Metamorphic Testing 20 Years Later: A Hands-on Introduction. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 538–539. https://doi.org/10.1145/

3183440.3183468

[48] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data (SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

[49] Andreas Seltenreich. 2019. SQLSmith. https://github.com/anse1/sqlsmith

[50] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. 618–622.

[51] SQLite3. 2020. Generated Columns. https://sqlite.org/gencol.html

[52] SQLite3. 2020. How SQLite Is Tested. https://www.sqlite.org/testing.html

[53] SQLite3. 2020. Most Widely Deployed and Used Database Engine. https://www.sqlite.org/mostdeployed.html

[54] SQLite3. 2020. The SQLite Query Optimizer Overview. https://www.sqlite.org/optoverview.html

[55] SQLite3. 2020. The Use Of assert() In SQLite. https://www.sqlite.org/assert.html

[56] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1493–1509. https://doi.org/10.1145/3318464.3386134

[57] Tencent Blade Team. 2019. Magellan 2.0. https://blade.tencent.com/magellan2/index_en.html

[58] Manasi Vartak, Venkatesh Raghavan, and Elke A. Rundensteiner. 2010. QRelX: Generating Meaningful Queries That Provide Cardinality Assurance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 1215–1218. https://doi.org/10.1145/1807167.1807323

[59] Marianne Winslett and Vanessa Braganholo. 2019. Richard Hipp Speaks Out on SQLite. *SIGMOD Rec.* 48, 2 (Dec. 2019), 39–46. https://doi.org/10.1145/3377330.3377338

[60] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a Learning Optimizer for Shared Clouds. *Proc. VLDB Endow.* 12, 3 (Nov. 2018), 210–222. https://doi.org/10.14778/3291264.3291267

[61] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532