Keep It Simple: Testing Databases via Differential Query Plans

JINSHENG BA, National University of Singapore, Singapore
MANUEL RIGGER, National University of Singapore, Singapore

Query optimizers perform various optimizations, many of which have been proposed to optimize joins. It is pivotal that these optimizations are correct, meaning that they should be extensively tested. Besides manually written tests, automated testing approaches have gained broad adoption. Such approaches semi-randomly generate databases and queries. More importantly, they provide a so-called *test oracle* that can deduce whether the system's result is correct. Recently, researchers have proposed a novel testing approach called *Transformed Query Synthesis* (*TQS*) specifically designed to find logic bugs in join optimizations. *TQS* is a sophisticated approach that splits a given input table into several sub-tables and validates the results of the queries that join these sub-tables by retrieving the given table. We studied *TQS*'s bug reports, and found that 14 of 15 unique bugs were reported by showing discrepancies in executing the same query with different query plans. Therefore, in this work, we propose a simple alternative approach to *TQS*. Our approach enforces different query plans for the same query and validates that the results are consistent. We refer to this approach as *Differential Query Plan* (*DQP*) testing. *DQP* can reproduce 14 of the 15 unique bugs found by *TQS*, and found 26 previously unknown and unique bugs. These results demonstrate that a simple approach with limited novelty can be as effective as a complex, conceptually appealing approach. Additionally, *DQP* is complementary to other testing approaches for finding logic bugs. 81% of the logic bugs found by *DQP* cannot be found by *NoREC* and *TLP*, whereas *DQP* overlooked 86% of the bugs found by *NoREC* and *TLP*. We hope that the practicality of our approach—we implemented in less than 100 lines of code per system—will lead to its wide adoption.

CCS Concepts: • **Information systems → Query optimization**; • **Security and privacy → Database and storage security**.

Additional Key Words and Phrases: Join, logic bug

ACM Reference Format:
Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 188 (June 2024), 26 pages. https://doi.org/10.1145/3654991

1 INTRODUCTION

A key feature of relational Database Management Systems (DBMSs) is to join data in multiple tables using a `JOIN`. Various strategies and optimizations have been proposed to optimize the execution of joins [14, 15, 34]. Given the complexity of such optimizations, query optimizers might apply a semantically incorrect optimization, which could result in incorrect results being produced. These errors are called logic bugs. It is challenging to find logic bugs as they silently produce incorrect results—unlike, for example, crash bugs [51, 52], which cause the process to be terminated. In Listing 1, the second query at line 8 triggers a logic bug in the DBMS, as it should return a non-zero result, instead of zero.

Recently, automated testing approaches for DBMSs have gained broad adoption to find logic bugs [26, 41–43], as they can often find many bugs that have been overlooked by manually written

Authors' addresses: Jinsheng Ba, National University of Singapore, Singapore, bajinsheng@u.nus.edu; Manuel Rigger, National University of Singapore, Singapore, rigger@nus.edu.sg.

ACM 2836-6573/2024/6-ART188
https://doi.org/10.1145/3654991

Listing 1. A bug found that may incur money loss by Differential Query Plans (*DQP*) in MySQL.

```
CREATE TABLE user(user_id DECIMAL PRIMARY KEY);
CREATE TABLE transaction(transitition_id TEXT, amount DECIMAL(10,2) NOT NULL);
CREATE INDEX i0 ON transaction(transitition_id(5));
INSERT INTO user VALUES(1), (2);
INSERT INTO transaction VALUES('1_c12934', 100000), ('1_e3b664', -10);

SELECT IFNULL(SUM(amount), 0) AS balance FROM user JOIN transaction ON
    transaction.transitition_id = user.user_id;  -- 99990.00 ✔
SELECT /*+ JOIN_ORDER(transaction, user)*/ IFNULL(SUM(amount), 0) as balance
    FROM user JOIN transaction ON transaction.transitition_id = user.user_id;
    -- 0.00 🐞
--------------------------------------------------------------------------------
nested_loop                              nested_loop
+- table                                 +- table
|  table_name: user                      |  table_name: transaction
|  access_type: index                    |  access_type: all
+- table                                 +- table
|  table_name: transaction               |  table_name: user
|  access_type: all                      |  access_type: eq_ref
```

tests, which are costly to write. Importantly, such approaches provide so-called *test oracles*, mechanisms to check whether the computed result by the DBMS is correct. Test oracles are typically either combined with semi-random database and query generators [3, 50], or existing benchmarks such as TPC-H [33] or TPC-DS [48]. *TQS* [46] is an automated testing approach for detecting logic bugs in query optimizations. Notably, it is the state-of-the-art approach for testing join optimizations. To tackle the test-oracle problem, it simulates joins to derive a query's ground-truth results, and the simulation is performed by table splitting. Specifically, it validates the correctness of join optimizations by splitting a given table into several sub-tables, and deriving ground-truth results of a query that joins these sub-tables by retrieving the given table. To generate more diverse test cases for finding more bugs, *TQS* randomly injects noise, such as `NULL` and `0`, to these sub-tables and models database schemas as a graph data model to evaluate the similarity of `JOIN`s. 115 bugs were found by this approach as claimed in the *TQS* paper. However, *TQS* suffers from two major challenges. First, this method is complex to understand and implement. *TQS* requires splitting and maintaining the data schemas with reference to a given table and modeling data schemas into graphs to decide whether two graphs are isomorphic for evaluating the similarity of queries. Second, the testing scope is small. *TQS* can apply only to equijoins. Although, as claimed in the *TQS* paper, this method could conceptually be extended to non-equijoins, this method cannot test other SQL features, which are directly executed to obtain results in *TQS*.

To understand the bug-finding effectiveness of *TQS*, we studied the bug reports of *TQS* in the public issue trackers. We identified 15 unique bugs, of which 14 were reported by showing that executing the same query with different query hints produces different results, as illustrated in Listing 1. Thus, deriving the ground-truth results is not necessary for finding these bugs.

Based on our observation, in this paper, we propose a simple and easy-to-understand approach to achieve the same level of bug-finding effectiveness as *TQS*. We propose checking the result consistency of executing different query plans of the same query, and refer to this method as *Differential Query Plan* (*DQP*) testing. More formally, given a database D and a query Q, the DBMS

executes Q on D using query plan P to obtain the result $Q(P, D)$. For another possible query plan P' for Q, $Q(P, D) \neq Q(P', D)$ indicates a bug.

To realize this technique as a black-box approach that eschews modifications to the DBMSs, we propose using query hints and setting system variables that are already provided by the DBMSs to affect the generated query plans. We believe that this technique is obvious and simple, but addresses both challenges of *TQS* and has a similar bug-finding effectiveness as *TQS*. Moreover, *DQP* can test more query optimizations rather than only join optimizations, as query hints and system variables can affect the optimizations of other SQL features. Importantly, the approach is straightforward to implement, as *DQP* does not need to maintain data structures, such as graphs and sub-tables, for deriving the ground-truth results.

Listing 1, which we briefly introduced above, shows a motivating example of a bug found by *DQP* in MySQL. Suppose we are in a bank scenario in which MySQL stores the user information in the table `user` and transaction records in the table `transaction`. Lines 1–3 create both tables with an index, and lines 4–5 insert data into both tables. The data in column `transaction_id` is composed of a user ID and a randomly generated transaction ID; for example, `1_c12934` represents the user `1` making a transaction whose ID is `c12934`. Line 7 checks the balance of user `1` and obtains the expected result, `9990.00`. If the query results in an inefficient query plan, a database administrator might decide to enforce another query plan by adding a query hint as shown in line 8. The hint `/*+ JOIN_ORDER(transaction, user)*/` instructs the DBMS to process table `transaction` before `user` when performing the join. However, this query returns a wrong result `0.00`. Both query plans are shown in lines 10–16. This bug may incur severe consequences, as all money of user `1` is lost.

We implemented *DQP* in less than 100 lines of Java code for each DBMS based on *SQLancer* [42], a widely used DBMS testing framework. *SQLancer* provides generators for databases and queries that we reused. We evaluated *DQP* on three DBMSs used in the *TQS* paper, MySQL, MariaDB, and TiDB. The results show that *DQP* can reproduce 14 out of the 15 unique bugs found by *TQS*, and all 10 bugs related to join optimizations. Although these systems were extensively tested by *TQS*, *DQP* found 26 previously unknown unique bugs, and 21 of them are logic bugs. Of these logic bugs, 15 are related to join optimizations, suggesting that these bugs were overlooked by *TQS*, and 6 are related to other query optimizations, meaning that they cannot be found by *TQS*. Compared with *TQS*, *DQP* is simple yet general, and efficient. We also compared the bugs found by *DQP* with the testing approaches *Non-optimizing Reference Engine Construction* (*NoREC*) [41], and *Ternary Logic Partitioning* (*TLP*) [42]. both of which were proposed as general bug-finding techniques for logic bugs with no focus on finding join optimization bugs. *NoREC* and *TLP* cannot find 17 of 21 logic bugs found by *DQP*, and *DQP* cannot find 35 of 40 logic bugs found by *NoREC* and *TLP*. In summary, *DQP* is not only a simple alternative to *TQS*, but also complementary to *NoREC* and *TLP*.

Overall, we make the following contributions:

- We studied the bug-finding effectiveness of a state-of-the-art work *TQS* for finding logic bugs in join optimizations;
- We demonstrate that the simple and easy-to-understand testing approach *Differential Query Plans* (*DQP*) testing shows the same level of bug-finding effectiveness as the more complex approach *TQS*;
- We implemented and evaluated the approach, which has found 26 unique, previously unknown bugs in widely-used DBMSs. The source code of *DQP* is publicly available, and has been integrated into *SQLancer*.[1]

[1] https://github.com/sqlancer/sqlancer/issues/918

2 *TQS* STUDY

TQS successfully found bugs in MySQL, MariaDB, TiDB, and PolarDB. However, it is a complex method that requires implementing multiple graphs and tables as internal components for deriving the ground-truth results. In this section, we study *TQS* to understand whether these bugs found by *TQS* can be found by a simpler method by answering the following questions:

RQ.1 Join-related Bugs. How many bugs reported by *TQS* are related to join optimizations? *TQS* aims to find bugs in join optimizations, so we study how many found bugs are related to them.

RQ.2 Bug Justifications. How were the bugs reported by *TQS*? Convincing developers that their DBMS, and not *TQS*, is computing an incorrect result might be challenging. As detailed in our answer to this question, we found that the bugs were explained not based on their ground-truth results, which motivates our simpler testing approach.

2.1 *TQS* Summary

Transformed Query Synthesis (*TQS*) [46] was proposed to detect logic bugs in join optimizations. It includes two major components: *Data-guided Schema and query Generation* (*DSG*) and *Knowledge-guided Query space Exploration* (*KQE*). *TQS* requires a wide table as an input table. The input table can be manually given, and in the *TQS* paper, the authors used the TPC-H[2] and KDD Competition[3] databases. First, *DSG* splits the wide table into multiple sub-tables through database normalization, which is an established technique that minimizes data redundancy and dependency by organizing data into separate tables. Then, *DSG* randomly constructs a query to join these sub-tables. The ground-truth results are derived by retrieving the wide table. The derivation process is not easy to implement as maintaining the relations between the given wide table and the split tables is necessary and complex. To make the generated queries more diverse, *KQE* evaluates whether a randomly generated query is similar to a previous query, and will adjust the random generation process to reduce the possibility of generating similar queries. The similarity is evaluated by modeling database schemas as an embedding-based graph, in which each query is a sub-graph, and *KQE* checks whether two sub-graphs are isomorphism. The authors claimed that *TQS* found 115 bugs within 24 hours including 7, 5, 5, and 3 types of bugs in MySQL, MariaDB, TiDB, and PolarDB.

2.2 Study Scope

We chose the public bug list from *TQS*[4] as our studied target. The public bug list is the only source that we could obtain to study *TQS* except for its paper. We did not involve the source code of *TQS*, because the source code is unavailable and the authors answered in emails that they were currently not able to provide it to us. We explain our attempts to obtain the source code in Section 6.

2.3 Data Preprocessing

Target DBMSs. We studied the bug reports of MySQL, MariaDB, and TiDB. *TQS* was originally evaluated on four DBMSs: MySQL, MariaDB, TiDB, and PolarDB, but we observed that the public bug list does not include the bug reports of PolarDB. As a result, we studied the bug reports of the first three DBMSs, in which the authors claimed that *TQS* found 92 bugs of 17 bug types. While the paper claims that all found bugs had been reported, the actual number of bug reports in the public bug list is 21, namely 11 bug reports in MySQL, 5 in MariaDB, and 5 in TiDB.

[2]https://www.tpc.org/tpch/

[3]https://archive.ics.uci.edu/dataset/129/kdd+cup+1998+data

[4]https://github.com/xiutangzju/tqs/blob/d5f8f5/index.md

Table 1. The bugs reported by *TQS*.

DBMS	Bug	Type ID	Unique	Join	Query Plan
MySQL	106713	3	✓		✓
MySQL	106715	4	✓	✓	✓
MySQL	106716	7	✓	✓	✓
MySQL	106717	5	✓		✓
MySQL	106718	2	✓		✓
MySQL	106611	6			✓
MySQL	106710	1	✓		✓
MySQL	99273		✓		
MySQL	109211		✓	✓	✓
MySQL	109212		✓	✓	✓
MariaDB	28214	8	✓	✓	✓
MariaDB	28215	9	✓	✓	✓
MariaDB	28216	10	✓	✓	✓
MariaDB	28217	11	✓	✓	✓
MariaDB	29695	12	✓	✓	✓
TiDB	33039	13		✓	✓
TiDB	33041	14		✓	✓
TiDB	33042	15	✓	✓	✓
TiDB	33045	16		✓	✓
TiDB	33046	17		✓	✓

To avoid that we missed any bug reports, as the list of found bugs provided by the *TQS* authors could be incomplete, we further searched the submission history of the first author in the corresponding issue trackers. We did not search for other authors as we could not find other authors' accounts in these issue trackers. Specifically, we searched the issue trackers of MySQL,[5] MariaDB,[6] and TiDB.[7] We failed to identify other bug reports. We show all bug reports in Table 1, in which we excluded bug #106473, because the developers of MySQL rejected the bug report.[8] We also observed that bugs #106611, #106710, and #99273 were reported by a non-author, which is mentioned in the acknowledgment of the *TQS* paper. Based on our observation and investigation, we infer that the 92 bugs in the paper refer to bug-inducing test cases, a large portion of which are duplicates, instead of unique, valid bugs.

Bugs in the TQS paper. To validate our assumption that 92 bugs in the *TQS* paper refer to bug-inducing test cases, we did a matching analysis to examine the correlation between the public bug list and the 17 bug types from Table 4 in the *TQS* paper. Specifically, for a bug in the public bug list, we searched for a matching bug type with the same bug status, bug severity, and similar description described in the *TQS* paper. We observed that the titles of bug reports are similar to the descriptions of bug types, but not exactly the same, so we adopted the algorithm gestalt pattern matching [39] to calculate whether two strings are similar, and the similarity is indicated as a floating number ranging from 0 to 1 indicating the degree of similarity. We deemed the highest score as the closest

[5]https://bugs.mysql.com/search.php?cmd=display&status=All&severity=all&reporter=16399198
[6]https://jira.mariadb.org/browse/MDEV-29695?jql=reporter="XiuTang"
[7]https://github.com/pingcap/tidb/issues?q=is:issue+author:xiutangzju
[8]https://bugs.mysql.com/bug.php?id=106473

match. After matching, we further manually examined and corrected the matching according to the semantic information of bug reports. Specifically, we matched bug #28217 and bug type 11 as they have similar descriptions—incorrect results by limiting join buffers but different severity levels, although they have different bug severity. We also matched bug #33042 and bug type 15 as they have the same keyword "empty resultset".

In Table 1, the column *Type ID* shows our matching results. 17 bug reports can be matched to the 17 bug types in the *TQS* paper, and 3 bug reports have no matching bug types. This result shows that each bug report in the public bug list refers to a bug type in the *TQS* paper, rather than indicating a bug. Three bug reports have no matched bug types, and a possible explanation is that they were submitted after submitting the *TQS* paper.

Unique bugs. Given the 20 bug reports, we investigated the uniqueness of these bugs according to the developers' responses. Typically, developers clearly respond to a bug report if it duplicates a previously reported bug. We carefully reviewed all developers' responses in the bug reports to check whether a bug is a duplicate.

In Table 1, the column *Unique* shows the unique bugs. 15 of the 20 bugs are unique. For MySQL, bug #106611 is a duplicate of the previously found bug #105773, and the developers confirmed this duplication within 24 hours after submitting the bug report. For TiDB, bugs #33049, #33041, #33045, #33046 are duplicates of bug #33042 reported by *TQS* as well, and these duplicates were confirmed by developers within 3 days after submitting the bug reports. We further study *TQS* based on the 15 unique bugs.

RQ.1 Join-related Bugs

We evaluated how many bugs are related to join optimizations. *TQS* aims to detect logic bugs in join optimizations, *DSG* derives ground-truth results of a join, and *KQE* drives the test case generation towards exercising diverse join optimizations. Therefore, it is important to determine how many bugs are related to join optimizations. We examined the test cases in the bug reports and checked whether a test case includes at least one **`JOIN`** clause. If so, we deemed the bug report to be related to join optimizations. Although join optimizations might also apply to other clauses, such as subqueries [13], *DSG* cannot derive the ground-truth results for these clauses,[9] so we considered only queries with **`JOIN`** clauses as join-related queries for this study.

In Table 1, the column *Join* shows the bug reports whose test case includes at least one **`JOIN`** clause. In total, 10 of 15 unique bugs (67%), are related to join optimizations. For the five bugs that are not related to join optimizations, #106713, #106717, #106718, #106710, and #99273, a common feature is that their bug-inducing test cases include at least one **`SUBQUERY`** clause. The core components of *TQS* show no obvious contribution to the finding of these non-join-related bugs. It is unclear how *TQS* constructs the ground-truth results for these cases, because *TQS* directly executes non-join SQL clauses to obtain the results.[10]

RQ.2 Bug Justifications

We examined the bug descriptions and test cases of these bug reports to study how these bugs were reported and justified. We observed that all 10 join-related bugs and 14 of 15 unique bugs were reported in the same manner, by demonstrating that different query plans of the same query compute inconsistent results. Listing 2 shows an example of bug #106713 in MySQL. The author argued the buggy behavior by showing that a query with the query hint `/*+ no_semijoin()*/` returns a different result than the same query, but without the query hint. A query hint instructs the DBMS

[9] In Section 3.3 of *TQS* paper, the authors claimed: "*DSG randomly generates other expressions based on the join clauses.*"
[10] In Section 3.4 of *TQS* paper, the authors claimed: "*DSG also executes the generated filters and projections defined in the AST*".

Listing 2. MySQL bug #106713 found by *TQS*.

```
CREATE TABLE IF NOT EXISTS t0(c0 DECIMAL ZEROFILL  COLUMN_FORMAT DEFAULT);
INSERT HIGH_PRIORITY INTO t0(c0) VALUES(NULL), (2000-09-06), (NULL);
INSERT INTO t0(c0) VALUES(NULL);
INSERT DELAYED INTO t0(c0) VALUES(2016-02-18);

SELECT t0.c0 FROM t0 WHERE t0.c0 IN (SELECT t0.c0 FROM t0 WHERE (t0.c0 NOT IN
    (SELECT t0.c0 FROM t0 WHERE t0.c0 )) = (t0.c0) ); --
    {0000001985},{0000001996}
SELECT t0.c0 FROM t0 WHERE t0.c0 IN (SELECT /*+ no_semijoin()*/ t0.c0 FROM t0
    WHERE (t0.c0 NOT IN (SELECT t0.c0 FROM t0 WHERE t0.c0 )) = (t0.c0) ); --
    empty set
```

Listing 3. MySQL bug #99273 found by *TQS*.

```
CREATE TABLE t1 (a INT, b INT);
INSERT INTO t1 VALUES (1,1),(2,1),(3,2),(4,2),(5,3),(6,3);

SET SQL_MODE = 'ONLY_FULL_GROUP_BY';
SELECT a FROM t1 as t1 GROUP BY a HAVING (SELECT t1.a FROM t1 AS t2 GROUP BY b
    LIMIT 1); -- {1},{2},{3},{4},{5},{6}
INSERT INTO t1 values (null, 4);
SELECT a FROM t1 as t1 GROUP BY a HAVING (SELECT t1.a FROM t1 AS t2 GROUP BY b
    LIMIT 1); -- empty set
```

to generate or avoid a specific query plan, and `no_semijoin()` disables the semijoin during query optimization. The only exception is bug #99273, as shown in Listing 3. This bug is included in the public bug list, but can not be matched to any bug type in the *TQS* paper. This bug was explained by the unexpected behavior that a query returns fewer rows after inserting a row with **NULL**. The root reason is an incorrect optimization for **SUBQUERY**, but is not related to **JOIN**. It is unclear how *TQS* derives the ground-truth result for a **SUBQUERY**, as *TQS* can only derive the results of **JOIN**. We also noticed that this bug was found in 2020, while all other bugs were found in 2022. Based on our observations, we assume that most of these bugs can be found by checking inconsistencies in executing the same query with different query plans, which is much simpler than *TQS*.

14 of the 15 unique TQS bugs and all 10 **JOIN**-related bugs were reported by showing discrepancies across the executions of different query plans of the same query.

3 APPROACH

We propose a simple approach, which we term *Differential Query Plans* (*DQP*) testing, to find bugs in join optimizations. Our core idea is to find bugs by comparing the results of the same query while enforcing different query plans for it. Compared to *TQS*, *DQP* does not require implementing graph and table structures for deriving ground-truth results. Moreover, *DQP* supports finding bugs in a variety of query optimizations instead of only in equijoin optimizations. Our key contribution is not the novelty of the approach, but the insight that a simple and easy-to-understand technique performs at the same level as a more sophisticated approach.

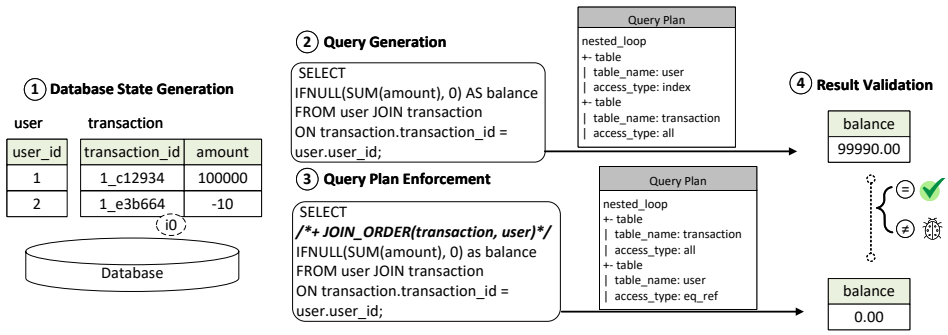

Fig. 1. Overview of *DQP*.

Approach overview. Figure 1 shows an overview of *DQP* illustrated based on Listing 1. First, *DQP* generates a database state D in step ①. Then, in step ②, *DQP* generates a query Q, and enforces different query plans P and P' to execute it. In step ③, *DQP* obtains the results of the executions. A discrepancy in the results, that is, $Q(P, D) \neq Q(P', D)$, indicates a potential bug.

Database State Generation (①). For a fully automated approach, we assume D to be randomly generated. Common generation methods include mutation-based methods [26, 52] and rule-based methods [41–43, 50]. To create D in Figure 1, *DQP* executes lines 1–5 in Listing 1. Generating a database state is not a contribution of this paper, and *DQP* can be paired with any database state generation method. In fact, D could also be manually specified.

Query Generation (②). Based on D, *DQP* randomly generates a query Q in step ② whose results we subsequently automatically validate to find bugs. Similar to database state generation, many query generation approaches have been proposed [5, 8, 23, 30, 37, 44, 45], and *DQP* can, in principle, be paired with any of these query generation methods.

Query Plan Enforcement (③). *DQP* executes Q, for which the DBMS derives a query plan P. Then, *DQP* attempts to force the DBMS to derive an alternative query plan P' for the same query. Query hints and system variables are two ways that affect query plans by using SQL keywords without the need to modify the source code of DBMSs. In Section 4 we describe more details about both ways. In Figure 1, *DQP* enforces a P' that has a different join order than P by the query hint `/*+ JOIN_ORDER(transaction, user)*/`.

Result Validation (④). In step ③, *DQP* executes $Q(P, D)$ and $Q(P', D)$ to obtain results, and we check their consistency. Here, $Q(P, D) = 99990.00$, while $Q(P', D) = 0.00$, so a bug is found.

4 IMPLEMENTATION

We implemented *DQP* in *SQLancer*,[11] a DBMS testing framework that randomly generates database states and queries complying with the SQL grammar, and subsequently refer to our prototype as *SQLancer+DQP*. We discuss the technical details for implementing *SQLancer+DQP* in this section.

4.1 Database and Query Generation for ①, ②

We adopted the grammar-based method provided by *SQLancer* to randomly generate syntactically correct SQL statements. *SQLancer* encodes the grammar of the DBMSs' SQL dialects, and *DQP*

[11] https://github.com/sqlancer/sqlancer

randomly walks the corresponding grammar tree to generate an SQL statement. To generate D, *DQP* generates non-query statements, such as `CREATE TABLE` and `CREATE INDEX`. Similarly, to generate Q, *DQP* randomly walks the tree to generate query statements, that is `SELECT`. Grammar-based generation methods are also used in *TQS* and SQLSmith [50].

Generating `JOIN`*s.* While *SQLancer* already generates `JOIN`s for many DBMSs, it lacks the support of `JOIN`s for MySQL and MariaDB. We updated *SQLancer* to support generating the `JOIN` clause for MySQL and MariaDB with reference to the code of `JOIN` in TiDB's implementation[12] in *SQLancer*.

4.2 Query Plan Enforcement for ③

Query hints and system variables are two ways that affect query plans by using SQL without requiring modifications to the source code of the DBMS under test.

Query hints. A query hint is a comment-like clause in a query and can affect the behaviors of the query optimizer. Query hints are widely supported by popular DBMSs, such as MySQL,[13] MariaDB,[14] and TiDB.[15] For the query hints that require table or column names as parameters, we randomly generate such names according to the query. In Figure 1, the query hint `/*+ JOIN_ORDER(transaction, user)*/` enforces the query optimizer to join both tables in a specific order, which is the difference between P' and P.

System variables. Another way to affect query plans is by setting system variables that affect the query optimizer. The variable `optimizer_switch` for MySQL[16] and MariaDB[17] is a system variable that affects query optimization and thus the generated query plans. Concretely, *DQP* executes a `SET` statement with the query to configure the system variable to enforce a different query plan. For example, in MariaDB, *DQP* may execute this `SET` statement and the query: `SET STATEMENT optimizer_switch='index_merge=on' FOR SELECT t0.c0 FROM t0`. The prefix `SET` configures the system variable taking effect for the following `SELECT` statement, and `index_merge` controls whether to enable the index merge optimization.

Efficiency consideration. For testing efficiency, we enforce multiple query plans $\{P', P'', \ldots\}$ by enumerating all possible query hints and values of the system variable in an iteration. Doing so is feasible as we observed that both query hints and potential values associated with the system variable are finite and small in number. We examined DBMS documents and extracted 32 query hints and 26 options for the system variable `optimizer_switch` in MySQL, 37 options for the system variable `optimizer_switch` in MariaDB, and 22 query hints in TiDB for enforcing different query plans. In Figure 1, for simplicity, we only show the executions of P and P'.

4.3 Result Validation for ④

We initially observed false alarms during step ④. As clarified by the DBMS developers, these false alarms were due to ambiguous queries, which refer to queries whose results are not guaranteed to be consistent or predictable. To exclude these false alarms, we identify ambiguous queries by checking whether a different row order in tables affects the result. After implementing this technique, we observed no false alarms. After an iteration, *DQP* continues with step ① or ② to start a new iteration. Since generating D is relatively slow, *DQP* returns to step ② by default. Only after a fixed

[12] https://github.com/sqlancer/sqlancer/blob/cddff6/src/sqlancer/tidb/ast/TiDBJoin.java
[13] https://dev.mysql.com/doc/refman/8.0/en/optimizer-hints.html
[14] https://mariadb.com/kb/en/optimizer-hints/
[15] https://docs.pingcap.com/tidb/stable/optimizer-hints
[16] https://dev.mysql.com/doc/refman/8.0/en/switchable-optimizations.html
[17] https://mariadb.com/kb/en/optimizer-switch/

number of iterations, does *DQP* return to step ①. The number is configurable, and we configured the number to be 10,000, which was empirically determined to work well in prior work [3].

Ambiguous queries. Ambiguous queries may incur false alarms, as also observed in other DBMS testing approaches [41, 42]. Investigating and analyzing all categories of ambiguous queries is challenging and exceeds the scope of this paper. We discuss the ambiguous queries that we encountered in practice. One kind of ambiguous query[18] is including non-aggregated columns in a `SELECT` clause. A column that is not included in `GROUP BY` is a non-aggregated column. If we include non-aggregated columns in `SELECT`, some DBMSs return the non-aggregated column of a random row from each group. The other DBMSs, such as PostgreSQL, reject such ambiguous queries. Listing 4 shows a concrete example that we encountered when testing TiDB. For the test case in the top half, both queries retrieve the column `t0.c0`, which is not included in `GROUP BY`. The function `CAST` converts both 0.9 and 0.8 to 1, so both rows in `t0` will be in the same group, but both queries return different results as they return a random row of the group. These ambiguous queries cause false alarms in the validation step ④.

Algorithm 1 Ambiguous query identification

Input: query: Q, two query plans of Q: P P', database: D

```
ambiguous = false
Minimize(Q, D, P, P')
for D' in Permutation(D) do
    if Validate(P, P', Q, D') ≠ Validate(P, P', Q, D) then
        ambiguous = true
        break
    end if
end for
```

Output: *ambiguous*

Ambiguous query identification algorithm. Algorithm 1 shows our algorithm to identify ambiguous queries by checking whether a different row order in tables affects the validation result. First, to reduce the computational complexity, we minimize Q and D. A bug-inducing test case identified by step ③ typically includes hundreds of SQL statements to initialize database states and query results. To efficiently execute them multiple times for identifying ambiguous queries, we minimize each test case both using C-Reduce [40] and manually. Figure 1 includes the minimized test case that includes only two tables and four rows. Then, we permutate the rows in all tables. For each permutation D', if it affects the validation result, $Validate(P, P', Q, D') \neq Validate(P, P', Q, D)$, the query is ambiguous. In Listing 4, permutating the rows in `t0` incurs a different result for the second query, and the discrepancy disappears. *DQP* identifies and ignores this test case as the discrepancy likely indicates an ambiguous query.

Algorithm scalability. We believe that in practice, Algorithm 1 is feasible, as databases used to reproduce most bugs in existing works are small after minimization; for example, the average number of SQL statements in minimized bug-inducing test cases is 3.69 across 499 historical bugs found by *SQLancer*.[19] That most test cases can be reproduced with only small bug-inducing test cases has been observed in various testing works, such as for file systems [32], Java programs [2],

[18] https://docs.pingcap.com/tidb/v6.5/dev-guide-unstable-result-set
[19] https://github.com/sqlancer/bugs/blob/96cbb8/bugs.json

Listing 4. An unstable behavior identified by recondition.

```
1  CREATE TABLE t0(c0 FLOAT);
2  INSERT INTO t0 VALUES (0.9), (0.8);
3  CREATE INDEX i0 ON t0(c0);
4  SET @@sql_mode='';
5
6  SELECT t0.c0 FROM t0 GROUP BY CAST(t0.c0 AS DECIMAL); -- {0.8}
7  SELECT /*+ IGNORE_INDEX(t0, i0)*/t0.c0 FROM t0 GROUP BY CAST(t0.c0 AS
       DECIMAL); -- {0.9}
8
9  ----------------------------------------------------------------------------
10 CREATE TABLE t0(c0 FLOAT);
11 INSERT INTO t0 VALUES (0.8), (0.9);
12 CREATE INDEX i0 ON t0(c0);
13 SET @@sql_mode='';
14
15 SELECT t0.c0 FROM t0 GROUP BY CAST(t0.c0 AS DECIMAL); -- {0.8}
16 SELECT /*+ IGNORE_INDEX(t0, i0)*/t0.c0 FROM t0 GROUP BY CAST(t0.c0 AS
       DECIMAL); -- {0.8}
```

and answer-set programs [35], and is known as the so-called *small-scope hypothesis*. For Figure 1, each table includes two rows, so the number of permutations is $2! * 2! = 4$. Except for the original permutation, the loop at line 3 executes at most 3 times. Through the test case minimization, the execution number of the loop reduces exponentially.

5 EVALUATION

To evaluate the effectiveness and efficiency of *DQP*, we sought to answer the following questions:

Q.1 Bug Reproduction. Can *DQP* find the bugs found by *TQS*?
Q.2 New Bugs. Can *DQP* find previously unknown bugs?
Q.3 Bug-finding Efficiency. How efficiently can *DQP* find bugs?
Q.4 Bug-finding Effectiveness. How effective is *DQP* compared to other test oracles for finding logic bugs?
Q.5 Coverage. To what extent does *DQP* cover query optimizers?

Tested DBMSs. We tested the same DBMSs, MySQL, MariaDB, and TiDB as we studied in Section 2. MySQL is one of the most popular relational DBMSs. MariaDB is another popular DBMS that was forked from MySQL. TiDB is a popular enterprise-class DBMS, and its open version on GitHub has been starred more than 35k times. Importantly, these DBMSs were also tested by *TQS*. Because the bug reports of PolarDB were not published by the *TQS* authors, we did not test PolarDB. For Q2 and Q4, we used the latest available development versions (MySQL: 8.1.0, MariaDB: 11.1.2, TiDB: 7.4.0). For a fair comparison in Q3, we used the same versions as *TQS* used (MySQL: 8.0.28, MariaDB: 10.8.2, TiDB: 5.4.0). All DBMSs were running in default configurations.

Experimental infrastructure. We conducted all experiments on an AMD EPYC 7763 processor that has 64 physical and 128 logical cores clocked at 2.45GHz. Our test machine uses Ubuntu 22.04.2 with 512 GB of RAM, and a maximum utilization of 60 cores.

Q.1 Bug Reproduction

We evaluated whether *DQP*, as a simple testing approach, can find the logic bugs found by *TQS*. As found in Section 2, 14 of 15 unique bugs and all 10 join-related bugs were reported in the same manner, which is similar to the approach of *DQP*, so we assumed that *DQP* could find these bugs as well. We used the test cases in the public bug reports from *TQS* (see Table 1) as the initial database state and the original query, and, following *DQP*, enforced a different query plan for the query as shown in step ③ in Figure 1. If we observe any discrepancy between the results returned by the original and derived queries, we deem that the bug can be found by *DQP*.

Results. *DQP* can identify 14 of 15 unique bugs that were reported by *TQS*. In Listing 2, which shows a previous-discussed bug-inducing test case found by *TQS*, the bug-inducing test case includes two queries, the only difference of which is the query hint, and the description of the bug reason is "`no_semijoin` produce wrong results". Therefore, *DQP* can derive the second query by adding the query hint `no_semijoin`, and easily find this bug.

We also found that all 10 join-related bugs can be found by *DQP*, as they were all reported in a manner like Listing 2. Although *TQS* finds a bug by comparing the execution result against a ground-truth result, the *TQS* authors explained the bugs to developers by providing a reference query whose result differed from the buggy query. The result demonstrates that *DQP* can find the majority of bugs that were found by *TQS*.

14 of 15 unique bugs, and all 10 join-related bugs found by *TQS* can be detected by *DQP*.

Q.2 New Bugs

Apart from reproducing existing bugs found by *TQS*, we evaluated whether *SQLancer+DQP* can find previously unknown bugs. We would expect so due to the broader testing scope. *DQP* can be applied also to non-equijoin and queries without **`JOIN`**s, given that these queries' query plans can be influenced by query hints or system variables. We ran *SQLancer+DQP* twice for 24 hours on three DBMSs aiming to find bugs. To reduce the possibility of finding duplicate bugs, between two runs, we disabled the query hints and system variables that contributed to the bugs found in the first run. Developers typically confirmed our bug reports within several days; however, these bugs usually required several weeks or months to be fixed for the next release version. Developers typically explicitly responded when a duplicate issue was reported. To avoid reporting duplicate bugs, we reported only the bugs that were likely unique, rather than all bug-inducing test cases. Specifically, for each query hint and available option of the system variables, we reported at most one bug.

Bug overview. Table 2 shows the unique, previously unknown 26 bugs found by *SQLancer+DQP*. The column *Logic* represents whether the bug is a logic bug, and the column *Join* represents whether the bug is related to join optimizations. We submitted 32 bug reports to the developers, of which 26 bugs were confirmed as unique and previously unknown bugs, 1 bugs were duplicates, 1 was waiting for further analysis, and 4 bugs were false alarms due to ambiguous queries. These false alarms inspired us to design Algorithm 1, and we observed no false alarms after implementing the algorithm. Note that #47019 and #47020 in TiDB are potential duplicates as they cannot be observed after fixing bug #46601. We are awaiting the developers' response to confirm whether they are duplicates.[20] Therefore, we deemed them unique as developers did not claim they were duplicates.

Logic bugs. Out of the unique and previously unknown 26 bugs, 21 were logic bugs in query optimizations, as they were found due to inconsistent results returned by different query plans of

[20] https://github.com/pingcap/tidb/issues/47019#issuecomment-1734913792

Table 2. Previously unknown and unique bugs found by *DQP*.

DBMS	Bug	Status	Severity	Logic	Join
MySQL	112243	Confirmed	Non-critical	✓	✓
MySQL	112242	Confirmed	Serious	✓	
MySQL	112264	Confirmed	Serious	✓	✓
MySQL	112269	Confirmed	Serious	✓	✓
MySQL	112296	Confirmed	Non-critical	✓	✓
MariaDB	32076	Confirmed	Major	✓	
MariaDB	32105	Confirmed	Major	✓	✓
MariaDB	32106	Confirmed	Major	✓	✓
MariaDB	32107	Confirmed	Major	✓	✓
MariaDB	32108	Confirmed	Major	✓	✓
MariaDB	32143	Confirmed	Major	✓	✓
MariaDB	32186	Confirmed	Major	✓	✓
TiDB	46535	Confirmed	Major	✓	✓
TiDB	46538	Confirmed	Moderate		
TiDB	46556	Confirmed	Major		
TiDB	46580	Fixed	Critical	✓	✓
TiDB	46598	Confirmed	Major	✓	
TiDB	46599	Confirmed	Major	✓	
TiDB	46601	Fixed	Critical	✓	
TiDB	47019	Confirmed	Major	✓	
TiDB	47020	Confirmed	Major	✓	✓
TiDB	47286	Confirmed	Major	✓	✓
TiDB	47345	Confirmed	Critical	✓	✓
TiDB	47346	Confirmed	Major		
TiDB	47347	Confirmed	Major		
TiDB	47348	Confirmed	Moderate		
Sum:	26			21	15

the same query. The non-logic bugs were due to internal errors and crashes that can be exposed without comparing the results of executing different query plans.

Join-related bugs. 15 of 21 logic bugs relate to join optimizations as their minimized test case requires at least one `JOIN`. While *DQP* can find bugs across various query optimizations, the majority of the found bugs relate to join optimizations, which is the test target of *TQS*. To identify bugs related to join optimizations, we followed the same classification method as in Section 2, which considers join optimization bugs as logic bugs that include at least one `JOIN` clause. The results show that join optimizations are more buggy than other query optimizations, and *TQS* overlooked our found bugs in join optimizations. Our simple approach, *DQP*, shows a surprising effectiveness in finding these join-optimization bugs.

Bug severity. One important question is whether the bugs found by *DQP* are considered important by the developers. For TiDB, whose bug severity is specified by developers, 12 of 14 found bugs have the bug severity of *Major* or *Critical*, which represents that the bugs seriously affected the target system and typically have high priorities to be fixed. The one bug found by *TQS* also has the

Listing 5. Bug #112242 found by *DQP* by setting system variables in MySQL.

```
CREATE TABLE t0(c0 INT);
INSERT INTO t0(c0) VALUES(1);
CREATE INDEX i0 USING HASH ON t0(c0) INVISIBLE;

SELECT t0.c0 FROM t0 WHERE COALESCE(0.6) IN (t0.c0); -- {}
SET SESSION optimizer_switch = 'use_invisible_indexes=on';
SELECT t0.c0 FROM t0 WHERE COALESCE(0.6) IN (t0.c0); -- {1}
```

Listing 6. Bug #46580 found by *DQP* by setting query hints in TiDB.

```
CREATE TABLE t0(c0 INT);
CREATE TABLE t1(c0 BOOL, c1 BOOL);
INSERT INTO t1 VALUES (false, true);
INSERT INTO t1 VALUES (true, true);
CREATE VIEW v0(c0, c1, c2) AS SELECT t1.c0, LOG10(t0.c0), t1.c0 FROM t0, t1;
INSERT INTO t0(c0) VALUES (3);

SELECT COUNT(v0.c2) FROM v0,  t0 CROSS JOIN t1 ORDER BY -v0.c1; -- empty set
SELECT /*+ MERGE_JOIN(t1, t0, v0)*/COUNT(v0.c2) FROM v0,  t0 CROSS JOIN t1
    ORDER BY -v0.c1; -- {4}
```

bug severity of *Critical*. For MySQL and MariaDB, we found that the bug severities are specified by users and typically not updated by the developers. Thus, we believe that they are inaccurate. However, since they were reported in the *TQS* paper, we also provide them for comparison. 10 of the 12 found bugs in MySQL and MariaDB have bug severity of *Serious* and *Major*, while all 12 bugs in the *TQS* paper have the bug severity of *Serious*, *Major*, or *Critical*. To further demonstrate the importance of our found bugs, we present two selected examples.

Example 1: a bug found by setting system variables. Listing 5 shows bug #112242 we found in MySQL by controlling the system variable `optimizer_switch`. The configuration `use_invisible_indexes` controls whether the query optimizer considers invisible indexes, which are excluded from query optimizations by default. In this example, the index `i0` is set to `INVISIBLE`, so the first query retrieves the data without using the index. When setting the variable to `use_invisible_indexes=on`, the second query uses index `i0` to retrieve the data. This bug is due to an incorrect index optimization. Without *DQP*, it is difficult to know if a query using that index returns an incorrect result. We specified the severity *Serious* when submitting the bug report. Of the 21 logic bugs *DQP* found in query optimizations, 10 bugs are found by setting system variables.

Example 2: a bug found by setting query hints. Listing 6 shows bug #46580 we found in TiDB by setting the query hint `MERGE_JOIN`. The query hint `MERGE_JOIN` instructs the query optimizer to use the sort-merge join algorithm when executing the **`JOIN`** operator. Based on the developers' reply, the issue was in the `Projection` operation, which corresponds to a projection operation in relational algebra. When using `MERGE_JOIN`, the `Projection` operation incorrectly returns an empty output, so the first query returns an unexpectedly empty result. *DQP* found this bug by comparing the results of the same query with and without the query hint `MERGE_JOIN`. `Projection` is prevalent as it is usually executed for a **`SELECT`**, so the developers assigned the severity *Critical* to this bug and fixed it within one week. Of the 21 logic bugs *DQP* found in query optimizations, 11 bugs are found by setting query hints.

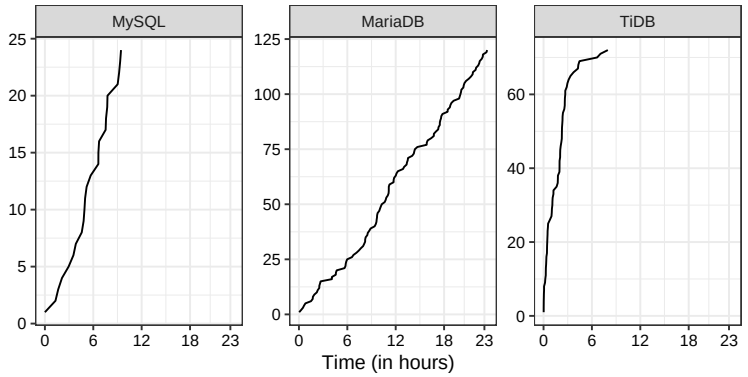

Fig. 2. Number of bug-inducing test cases found by *DQP* in 24 hours and 10 runs.

DQP enabled us to find and report 26 unique, previously unknown bugs missed by *TQS*.

Q.3 Bug-finding Efficiency

We evaluated how many bugs *DQP* can find in 24 hours. We ran *DQP* with the default configurations of *SQLancer* for 24 hours, and measured the number of bug-inducing test cases. We excluded bugs that cause crashes or internal errors, because they are not directly found by *DQP*, but by an implicit test oracle. Although *TQS* was evaluated in a similar experiment in Section 5.2 of the *TQS* paper, it is challenging to make a fair comparison, due to the aforementioned unavailability of its source code, and because some experimental configurations are unclear. First, both *TQS* and *DQP* adopt a grammar-based test case generation method, but the implementation differences are unclear, such as the possible expressions for `WHERE`. While other DBMS testing works [3, 52] also omit detailed descriptions, they provide the source code, from which this information can be extracted. Second, *TQS* supports multiple threads, but we have not found the number of threads used for their efficiency evaluation in Figure 8 of Section 5.2 in the *TQS* paper. For *SQLancer+DQP*, we ran 10 threads, which is a common practice for evaluating testing tools [25] and is also used by other DBMS testing work [3]. Third, it is unclear whether the *TQS* authors counted only logic bugs or all kinds of bugs. Last, *TQS* and *DQP* were evaluated on different machines, which have a significant impact on efficiency, so their efficiency results are not directly comparable.

Results. Figure 2 shows the number of bug-inducing test cases found by *DQP* in MySQL, MariaDB, and TiDB for 24 hours. In total, *DQP* found 24, 120, and 72 bug-inducing test cases in three DBMSs respectively. Due to several crash bugs found by *SQLancer+DQP*, MySQL and TiDB exited at around 9 hours. Compared with the results in Section 5.2 of the *TQS* paper, *DQP* demonstrates significant progress in bug detection efficiency compared to *TQS*. Recall that it is challenging to conduct a fair comparison with *TQS*. Nevertheless, the substantial number of bug-inducing test cases found by *SQLancer+DQP* demonstrates its efficiency even without sophisticated techniques to improve test-case generation.

SQLancer+DQP found 216 bug-inducing test cases in 24 hours in MySQL, MariaDB, and TiDB.

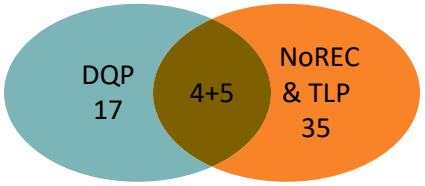

Fig. 3. The number of bugs detected by oracles.

Listing 7. Equivalent test cases of Listing 5 for *NoREC* and *TLP*.

```
CREATE TABLE t0(c0 INT);
INSERT INTO t0(c0) VALUES(1);
CREATE INDEX i0 USING HASH ON t0(c0) INVISIBLE;
-------------------------------------NoREC-------------------------------------
SELECT COUNT(*) FROM t0 WHERE COALESCE(0.6) IN (t0.c0); -- {0}
SELECT SUM(count) FROM (SELECT (COALESCE(0.6) IN (t0.c0)) IS TRUE AS count
    FROM t0) as t; -- {0}
--------------------------------------TLP--------------------------------------
SELECT t0.c0 FROM t0; -- {1}
SELECT t0.c0 FROM t0 WHERE COALESCE(0.6) IN (t0.c0) UNION SELECT t0.c0 FROM t0
    WHERE NOT (COALESCE(0.6) IN (t0.c0)) UNION SELECT t0.c0 FROM t0 WHERE
    (COALESCE(0.6) IN (t0.c0)) IS NULL; -- {1}
```

Q.4 Bug-finding Effectiveness

We compared *DQP* with two state-of-the-art oracles for finding logic bugs: Non-optimizing Reference Engine Construction (*NoREC*) [41], and Ternary Logic Partitioning (*TLP*) [42]. *NoREC* checks for inconsistent results of a predicate used in a query that the DBMS might optimize and one that is used in a query that is difficult to optimize. *TLP* expects a query and derives multiple more complex queries, each of which computes a partition of the result to check whether the combined partitions and the original query's results are equivalent. Both oracles are implemented in *SQLancer*. We did not consider other test oracles for finding logic bugs, such as Pivoted Query Synthesis (*PQS*) [43], which is not supported for the three evaluated DBMSs in *SQLancer*.

Methodology. We used the same methodology as prior works [21, 41, 42] to conduct a manual and best-effort analysis to identify the overlap and unique bugs found by *DQP*, *NoREC*, and *TLP*. It is difficult to distinguish whether two bug-inducing test cases found by different methods trigger the same underlying bug [28]. We considered only the minimized test cases that were reported to developers assuming that each test case represented a unique bug. While we cannot completely rule out misclassifications that might be due to overlooking that a bug could be found by another query, we believe that the majority of cases were clear. In total, we collected all 41 logic bugs, of which 40 are reproducible, found by *NoREC* and *TLP* for MySQL, MariaDB, and TiDB from the public bug list,[21] and 21 logic bugs found by *DQP* in Table 2. Then, we used a bug-inducing test case for *DQP* to derive another test case by applying *NoREC* and *TLP* to the same database and the corresponding query, and *vice versa*.

Results. Figure 3 shows the number of bugs found by *DQP*, *NoREC*, and *TLP*. 17 of 21 logic bugs found *DQP* cannot be found by *NoREC* or *TLP*. Out of the 17 bugs, 10 bugs are because both the

[21] https://github.com/sqlancer/bugs/blob/96cbb856/bugs.json

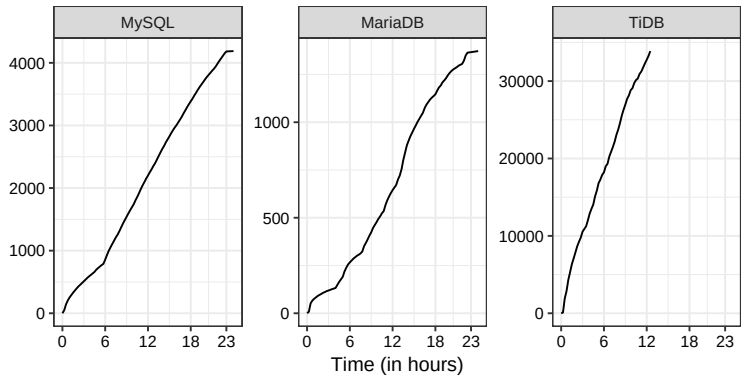

Fig. 4. Average number of unique query plans covered by *DQP* in 24 hours and 10 runs.

original query and the derived query result in correct or incorrect query plans. The other 7 bugs cannot be rewritten to equivalent test cases for *NoREC* or *TLP* due to the lack of necessary clauses for both oracles, such as **WHERE**. We also found that *DQP* cannot reproduce all 4 bugs found by *NoREC*, and 31 of 36 bugs found by *TLP*. The result shows that the bugs found by *DQP* rarely overlap with the bugs found by *NoREC* and *TLP*, suggesting that *DQP* is complementary to *NoREC* and *TLP*.

Example. Listing 7 shows an example of rewriting the bug-inducing test case of Listing 5 to equivalent test cases for *NoREC* and *TLP*—note that this is a mechanical transformation. For *NoREC*, the first query is the same as the original query in line 5 of Listing 5, and the second query is derived from the first query by moving the predicate in **WHERE**. For *TLP*, the first query is generated by omitting the **WHERE** in the original query, and the second query is a union of three queries with different predicates in **WHERE**. Both oracles find a bug if both queries return inconsistent results. Without setting the option `use_invisible_indexes=on`, the buggy index `i0` is not considered, so all queries checked by *NoREC* and *TLP* fail to use the buggy index required to expose the bug.

NoREC and *TLP* cannot find 17 of 21 logic bugs found by *DQP*.

Q.5 Coverage

We evaluated how well *DQP* exercises query optimizers, which is the key component that we aimed to test. We considered various metrics to capture the notion of coverage. First, since *DQP* enforces different query plans of the same queries, we examined how comprehensively query plans are covered by *plan coverage*, which refers to the ratio of exercised unique query plans to the estimated number of all observable unique query plans. Then, we used query hints and system variables to enforce query plans, so we evaluated to what extent they affect query optimizers by *hint and variable coverage* and *join coverage*. Last, we also evaluated *code coverage*, a common metric to evaluate how much code is tested.

Plan coverage. We measured the ratio of unique query plans *DQP* covers for all observable unique query plans. A query plan represents an optimized query, and a higher number of unique query plans implies that more query optimization strategies are applied. A challenge with respect to measuring the number of unique query plans is that query plans include unstable auxiliary information, which usually differs for almost every query plan. We consider a query plan *structurally unique*, if the query plan is still unique after removing such information. To exclude this information,

Table 3. The number of query hints or system variables that affect the three categories of query optimizations.

DBMS	Join	Index	Table
MySQL	14	26	18
MariaDB	18	5	14
TiDB	10	4	8
Sum:	42	35	40

we omitted schema names (*e.g.*, column and table names), estimated cost (*e.g.*, cardinalities), and random identifiers (*e.g.*, line identifiers) in query plans. This method follows the practice reported in a prior work [3]. Another challenge is the unknown upper bound of the number of query plans, as it is unclear how many possible combinations of operations for a query plan exist; note that the number is infinite in principle because an additional `JOIN` clause will typically result in a more complex query plan. As a best effort, we estimated the upper bound by combining all unique query plans covered by *DQP*, *NoREC*, and *TLP* across 24 hours and 10 runs, assuming the number of combined unique query plans as the upper bound. Suppose D_i, N_i, T_i represent the set of unique query plans covered by three oracles respectively for a DBMS in run i, then the estimated number of the upper bound is $|\bigcup_{i=1}^{10}(D_i \cup N_i \cup T_i)|$.

Results. Figure 4 shows the average number of unique query plans covered by *SQLancer+DQP* across 10 runs in 24 hours. In summary, for MySQL, MariaDB, and TiDB, the estimated upper bounds of plan coverage are 27156, 7553, and 253947, and *SQLancer+DQP* covers 15.42% (4187.5), 18.17% (1372.4), and 13.34% (33876.6) on average for each run. Due to several crash bugs found by *SQLancer+DQP* in TiDB, all 10 runs exited in around 12 hours. Although in a shorter time period, the most unique query plans were covered in TiDB. A possible reason is that TiDB includes richer elements in query plans than others. TiDB is a distributed DBMS, and, for example, indicates the execution node of each operator in query plans, while other DBMSs do not have similar information. Although achieving less than an average of 20% plan coverage for the three DBMSs, *DQP* achieves a much higher plan coverage than *NoREC* and *TLP*, both of which achieved less than 1% average plan coverage across 10 runs. This is expected, because *NoREC* and *TLP* do not optimize for this metric. We note that the overall low coverage can be explained by diverse query plans being explored across runs—less than 50% overlapped for *SQLancer+DQP* across 10 runs, and thus the sum of the average coverage numbers for each run of *DQP*, *NoREC*, and *TLP* is not close to 100%. The reason may be randomly generated databases and queries, which typically differ across runs. We cannot compare the plan coverage by *DQP* and *TQS*, because *TQS*'s source code is unavailable, and no query plan coverage numbers were reported in its paper. For all three DBMSs, *SQLancer+DQP* covers thousands of unique query plans, which shows that *SQLancer+DQP* is effective in testing query optimization.

Hint and variable coverage. We identified three categories of query optimizations that can be affected by query hints or system variables. Table 3 shows the number of query hints or systems variables of each category. Although query plans and query optimizations are DBMS-specific and not directly comparable, we found that the three DBMSs provide hints or variables to affect common categories of optimizations: *Join*, the algorithms and orders of joining two tables; *Index*, the algorithms and applicable range for indexes; and *Table*, the strategies to write and read tables, such as table caching for repeated queries and full table scan for small tables. As a concrete example, to affect join optimizations, query hint `HASH_JOIN` can be used in MySQL and TiDB to enforce the use

of the hash join algorithm, and variable `hash_join_cardinality` can be used in MariaDB to indicate whether historical cardinality statistics should be used for hash joins. Although MariaDB is derived from MySQL, both have a different number of query hints and system variables. For example, for the category *Index*, MySQL has two query hints and four system variables tailored for affecting the algorithm of index merge, which is an optimization for using indexes to merge results from multiple scans, while MariaDB does not have a similar hint or variable to affect it. TiDB has additional optimizations of the category *Table* for switching storage engines, which can be affected by query hints. For example, the query hint `READ_FROM_TIFLASH` is used to enforce reading tables from TiFlash, a storage engine of TiDB. This optimization is specific to TiDB, while MySQL and MariaDB can only specify storage engines when creating tables. All three DBMSs provide support for query hints and system variables, influencing the same three categories of query optimizations. However, they impact specific query optimizations to each DBMS.

Join coverage. Since *DQP* aims to find bugs in join optimization, we also evaluated how many join operators *SQLancer+DQP* covered in query plans by setting query hints and system variables across 10 runs in 24 hours. We examined whether our query plans cover the join operators illustrated in the documents of MySQL,[22] MariaDB,[23] and TiDB.[24] In total, both MySQL and MariaDB have 12 join operators, and TiDB has 3 join operators. *SQLancer+DQP* covered 7 out of 12 join operators for MySQL and MariaDB, and all 3 join operators for TiDB. For both MySQL and MariaDB, four join operators were not covered: `fulltext`, which is used for full-text indexes; `index_merge`, which is used for union or intersection expressions; `unique_subquery`, and `index_subquery`, both of which are used for subqueries. MySQL and MariaDB provide a specific query hint `INDEX_MERGE` to enable `index_merge`, but the join operator is not covered by our implementation of *DQP*, since it requires specific expressions in queries. We have not found any hint or variable that directly enforces the other three join operators. Additionally, MySQL's join operator `system` was not covered, which is used for system tables, and neither MariaDB's `ref_or_null`, which is used for index lookup with null values during joining. The reason for the two not-covered operators may be the randomness of test case generators, as either operator is covered by the other DBMS.

Code coverage. While we were primarily interested in the number of covered unique query plans, code coverage is a common metric for evaluating how much a system might be tested. We used gcov,[25] a coverage tool for C/C++ language, to collect line coverage of MySQL and MariaDB, and used cover,[26] a coverage tool for the Go language, to collect the statement coverage of TiDB. Line coverage is the default metric for gcov, and statement coverage is the default metric for cover. We ran *SQLancer+DQP* for 24 hours and 10 runs simultaneously. Due to resource limitations, each target DBMS ran one instance, and we measured their sum line and statement coverage across 10 runs. Since *TQS*'s source code is unavailable, we cannot compare the code coverage between *DQP* and *TQS*. Since *DQP* finds bugs in join optimization, we measured only the code coverage of query optimization. Specifically, we measured the code in the folder *sql* for MySQL and MariaDB, and in the folder *planner* for TiDB. The results show that *SQLancer+DQP* covered 22.2% and 27.7% line coverage for MySQL and MariaDB, and 36.1% statement coverage for TiDB. The coverage appears to be low, as less than 50% coverage for all DBMSs. However, this is expected because we cannot enumerate all possible query plans.

[22] https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_system
[23] https://mariadb.com/kb/en/explain/#type-column
[24] https://docs.pingcap.com/tidb/stable/explain-joins
[25] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[26] https://go.dev/testing/coverage/

DQP covers thousands of unique query plans and more than half join operators for MySQL, MariaDB, and TiDB in 24 hours.

6 DISCUSSION

We discuss some key characteristics of *DQP*, as well as the evaluation results of *TQS*.

Bug diversity. Our found bugs can affect a variety of different queries. The bugs were typically due to incorrect optimizations, such as the incorrect index optimization shown in Listing 5, and the incorrect join optimization shown in Listing 6. These buggy optimizations can affect other queries as well, not only the bug-inducing test cases that make use of specific query hints or values of system variables. For example, considering Listing 5, if the index `i0` is not created with `INVISIBLE` by `CREATE INDEX i0 USING HASH ON t0(c0)`, the first query `SELECT t0.c0 FROM t0 WHERE COALESCE(0.6)IN (t0.c0)` returns the incorrect result 1 without setting up any query hint or system variables.

Path to adoption. We believe that a simple testing approach has the potential for wide adoption. From a conceptual perspective, *DQP* is a general black-box approach that compares the results of different query plans, which is easy to understand. It is not necessary to instrument code for tracing internal execution information or understand how the result is computed. From an implementation perspective, *DQP* is easy to implement as we implemented *DQP* in less than 100 lines of Java code per DBMS. From an integration perspective, *DQP* can be paired with existing databases and query generators, or test suites. From an applicability perspective, *DQP* can test a significant number of DBMSs, as 8 out of 10 most popular relational DBMSs[27] support controlling query optimization by users. The remaining two DBMSs are Microsoft Access and Snowflake, for which we have not found any document that explicitly explains how to manually control query optimization. Considering these features of *DQP*, we argue that *DQP* can be widely adopted.

Contribution and novelty. The core contribution of this paper is that we demonstrated that the simple and easy-to-understand testing *DQP* approach shows the same level of bug-finding effectiveness as the more complex *TQS* approach. The authors of *TQS* mentioned the comparison of queries with query hints in Section 5.3 of the *TQS* paper by disabling the derivation of ground-truth results. Some systems in practice, such as DuckDB, already use similar techniques in their own testing framework as well. DuckDB does this by running both an unoptimized and optimized version of a query, and checking consistency of the results. It controls the optimizations by a specific statement `PRAGMA enable_verification`, which is specific to DuckDB.[28] Our core contribution is the insight that *DQP*, as a simple approach, achieves the same level of bug-finding efficiency as the sophisticated method *TQS*. In general, in a testing context, we believe that simple, practical approaches provide significant benefits over complex, but conceptually appealing ones.

The importance of TQS. TQS is the first approach for testing logic bugs in join optimizations, and thus demonstrated the severity of the problem. Importantly, *TQS* provides a new paradigm for finding logic bugs in DBMSs. In this work, we showed that a simple method achieves the same level of bug-finding efficiency as *TQS*. Nevertheless, *TQS* can find bugs that cannot be found by *DQP*. Bug #99273 in Table 1 was found without showing discrepancies across executing different query plans of the same query. Although it is unclear how *TQS* derives the ground-truth result for this query, as it lacks a `JOIN`, *DQP* cannot find this bug.

[27] https://db-engines.com/en/ranking/relational+dbms
[28] https://duckdb.org/dev/sqllogictest/intro#query-verification

Inconsistent bug number of TQS. We identified 15 unique bugs in the public bug list from *TQS*, while the authors of *TQS* claimed to have found 92 bugs. Based on our study, we suspect that the authors confused terminology by referring to *bug-inducing test cases* as "bugs" and *unique bugs* as "bug kinds", which we clarified in this work. We acknowledge that bug deduplication is an open problem [12], and we also observed duplicate issues being counted as bugs in other work.

Unavailable TQS source code. Two reasons prevent us from comparing with *TQS* in our study and evaluation. First, the authors of *TQS* have not released its source code. We sent two emails to all authors of *TQS* requesting the source code, but the authors replied that the source code was not ready for release: "*I'm currently working on a follow-up project that builds upon the research I presented. As a result, I'm in the process of refining and enhancing the codebase for both projects. Once this work is complete, I plan to make the source code available as open source or share it with colleagues who express an interest.*" We also noticed that the authors published another tool demonstration paper [47] that includes the implementation of *TQS*. Unfortunately, after carefully checking and debugging its source code,[29] we found that the repository lacks the core approach implementation of *TQS*, as also observed by another interested party.[30] Second, it is challenging to re-implement *TQS* as it consists of complex steps, with important details not being described in the paper. For example, the authors claimed "*We directly use these data-driven schema normalization methods to generate our testing database schema*", but it is not clear what concrete method they used to split the wide table and what sub-tables are generated. *TQS* adopts Abstract Syntax Tree (AST)-based random query generation, which is implemented "*similarly to RAGS and SQLSmith*", but it is insufficient to know what queries it can generate, such as what expressions are generated and the maximum depth of the AST. While it is understandable that these are not described in the paper, it prevented us from reimplementing the approach for comparison.

Threats to Validity. Our evaluation results face potential threats to validity. A major concern is that *TQS* source code and *TQS*'s bug reports on PolarDB are not available. To alleviate this risk, we communicated with the authors of *TQS*, who told us that *TQS*'s source code has not been ready for release yet. Because it is challenging to re-implement *TQS*, we extracted the public bug list and conducted a rigorous study described in Section 2. From a practical perspective, we compared the bugs found by *TQS* and *DQP* to evaluate their bug-finding capabilities. From a theoretical perspective, we discussed their conceptual differences in this section. Another concern is the correctness of our implementation. To mitigate this risk, we built *DQP* on a popular DBMS testing framework, *SQLancer*, and made the source code publicly available. The last concern is the reliability of the results we presented. To validate that our found bugs are real bugs, we reported each found bug to the DBMSs' developers and annotated the bug status according to developers' replies. We also made all bug reports public.

7 RELATED WORK

We briefly summarize the works that are most closely related to this work.

Finding optimization bugs. The query optimizer is one of the most critical components in DBMSs, so its reliability is important and has attracted much attention for testing. NoREC [41] detects logic bugs in query optimizations by checking whether the result of an optimized query equals an unoptimized query. Apart from logic bugs, performance bugs also exist in query optimizations. If a query optimization chooses an unexpectedly inefficient query plan, it can be considered a performance bug. Jung *et al.* proposed APOLLO [24], which compares the execution times of a

[29] https://github.com/xiutangzju/dlbd/tree/b85b1f
[30] https://github.com/xiutangzju/dlbd/issues/1

query on two versions of a database system to find performance regression bugs. Liu *et al.* proposed AMOEBA [27], which compares the execution time of a semantically equivalent pair of queries to identify an unexpected slowdown. Ba *et al.* proposed CERT [4] to find performance issues through testing cardinality estimation. In contrast, *DQP* specifically finds logic bugs in query optimizations focusing on join optimizations.

Manipulating query plans. Various techniques have been proposed to manipulate query plans. AEM [36] uses query hints to switch query plans for bypassing bugs in run-time. PgCuckoo [20] provides a plugin for PostgreSQL, so that PostgreSQL can execute arbitrary query plans. However, a significant challenge, as claimed in the PgCuckoo paper, is that manually manipulating query plans has a high invalidity rate as the operations in a query plan typically have dependencies on each other. TAQP [19] uses query hints to switch query plans and measures execution time to check whether the query plan chosen by query optimizers is the optimal one. Compared with these methods, *DQP* adopts a black-box manner to manipulate query plans by query hints and system variables for finding logic bugs.

DBMS fuzzing. Fuzzing is an efficient technique to find bugs in DBMSs, but aims to find security-relevant bugs, such as memory errors. SQLSmith [50], Griffin [16], DynSQL [22], and ADUSA [27] used grammar-based methods to generate test cases for finding memory errors. Squirrel [52], inspired by grey-box fuzzers such as AFL [49], used code coverage as guidance to generate diverse test cases for finding memory errors. Different from these methods, *DQP* aims to find logic bugs in DBMSs, especially in query optimizations.

Methodology. Several existing works in other domains adopt a similar methodology as this work to propose a simple technique that can outperform an existing sophisticated one. Kali [38] uses a simple method that only deletes functionality and outperforms previous sophisticated techniques for automatically generating software patches. Fu *et al.* [17] demonstrated that a simple tuned Support Vector Machine (SVM) can outperform a sophisticated Convolutional Neural Network (CNN) algorithm. This paper was also inspired by Kali.

Differential and metamorphic testing. Differential testing is a mature testing technique that compares the results of the same test case on different systems, and was proposed by McKeeman [29]. Metamorphic testing [10, 11] is a method that checks the correctness of a system by applying transformations to its input and examining the relationship between the original and transformed results. A significant difference between both methods is that metamorphic testing works on a single system and differential testing works on multiple systems. At a conceptual level, *DQP* can be classified as a metamorphic testing approach, in which we check whether a query returns the same result with different query plans. Existing metamorphic testing methods for testing DBMSs, such as TLP [42] and NoREC [41], which we mentioned above, apply predefined rules to modify predicates and move SQL clauses. Compared to these methods, *DQP* is simpler, as we only add query hints or set system variables.

Query generation. For query generation, two prominent approaches exist: targeted and random query generation. Targeted query generation, as proposed by Bati *et al.* [5], involves integrating execution feedback, such as code coverage, to guide the generation process toward a specific code location. Another method by Khalek *et al.* [1] employs a solver-backed technique to produce queries that are both syntactically and semantically correct. Given the computational complexity associated with generating queries that adhere to cardinality constraints, heuristic algorithms have been introduced [9, 31]. On the other hand, random query generation, exemplified by SQLsmith [50], employs a predefined grammar to stochastically generate queries that are semantically valid,

leading to the discovery of over 100 bugs in widely-used Database Management Systems (DBMSs). APOLLO [23] similarly relies on a predefined grammar for query generation to uncover regression performance issues. Importantly, *DQP* is adaptable to any query-generation methodology.

Database state generation. Similarly, for database state generation, two principal strategies are employed: targeted and random generation. Targeted database state generation, as demonstrated by QAGen [7], leverages symbolic execution to delineate constraints and subsequently generate queries that satisfy these constraints. SPQR [6] is another method that produces a database state corresponding to a given query and its anticipated outcomes. Random database state generation, as proposed by Gray *et al.* [18], introduces parallel algorithms to efficiently generate databases with billions of records. Coverage-based methodologies [26, 52] generate database states by modifying given SQL statements that were used in creating the original state. QPG [3] aims to generate diverse database states with the guidance of query plans. Similarly, *DQP* is amenable to any database state generation technique.

8 CONCLUSION

In this paper, we have studied the state-of-the-art testing approach for joins, *TQS*, and have proposed a simple, yet effective alternative approach, *DQP* testing. The core idea of *DQP* is comparing the consistency across the executions of different query plans of the same query, which we derive by adding query hints or setting system variables. Compared to *TQS*, *DQP* only needs to compare results instead of constructing multiple graphs and tables for deriving ground-truth results, and supports finding bugs in more query optimizations than equijoin optimizations. Our evaluation has demonstrated that *DQP* can find 14 of the 15 unique bugs and all 10 join-related bugs found by *TQS*. Additionally, *DQP* has found 26 previously unknown and unique bugs in MySQL, MariaDB, and TiDB, which were overlooked by *TQS*. As with *TQS*, *DQP* complements existing testing approaches, which find logic bugs also in other components than join optimization. Indeed, 81% of the logic bugs found by *DQP* cannot be found by *NoREC* and *TLP*, and *DQP* overlooked 86% of the bugs found by *NoREC* and *TLP*. *DQP* requires little implementation effort, is compatible with any test case generation methods, is similarly efficient as *TQS*, and is a black-box testing method. We encourage DBMS developers to use *DQP* in practice, as a cost-efficient way to find critical bugs in DBMSs.

ACKNOWLEDGMENTS

This research is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing <NRF-NCR25-Fuzz-0001>). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore.

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*. 329–332.

[2] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. 2003. Evaluating the "small scope hypothesis". In *In Popl*, Vol. 2.

[3] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2060–2071. https://doi.org/10.1109/ICSE48619.2023.00174

[4] Jinsheng Ba and Manuel Rigger. 2024. CERT: Finding Performance Issues in Database Systems Through the Lens of Cardinality Estimation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 917–917. https://doi.org/10.1145/3597503.3639076

[5] 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* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 1243–1251.

[6] Carsten Binnig, Donald Kossmann, and Eric Lo. 2006. Reverse query processing. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 506–515.

[7] 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* (Beijing, China) *(SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 341–352. https://doi.org/10.1145/1247480.1247520

[8] 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

[9] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. 2006. Generating queries with cardinality constraints for dbms testing. *IEEE Transactions on Knowledge and Data Engineering* 18, 12 (2006), 1721–1725.

[10] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. 2020. Metamorphic Testing: A New Approach for Generating Next Test Cases. *CoRR* abs/2002.12543 (2020). arXiv:2002.12543 https://arxiv.org/abs/2002.12543

[11] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–27.

[12] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 197–208.

[13] Mostafa Elhemali, César A Galindo-Legaria, Torsten Grabs, and Milind M Joshi. 2007. Execution strategies for SQL subqueries. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 993–1004.

[14] Pit Fender and Guido Moerkotte. 2013. Counter Strike: Generic Top-Down Join Enumeration for Hypergraphs. *Proc. VLDB Endow.* 6, 14 (2013), 1822–1833. https://doi.org/10.14778/2556549.2556565

[15] Pit Fender, Guido Moerkotte, Thomas Neumann, and Viktor Leis. 2012. Effective and Robust Pruning for Top-Down Join Enumeration Algorithms. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, Anastasios Kementsietsidis and Marcos Antonio Vaz Salles (Eds.). IEEE Computer Society, 414–425. https://doi.org/10.1109/ICDE.2012.27

[16] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin : Grammar-Free DBMS Fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 49:1–49:12. https://doi.org/10.1145/3551349.3560431

[17] Wei Fu and Tim Menzies. 2017. Easy over hard: A case study on deep learning. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 49–60.

[18] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J Weinberger. 1994. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*. 243–252.

[19] 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*. 1–6.

[20] Denis Hirn and Torsten Grust. 2019. PgCuckoo: Laying Plan Eggs in PostgreSQL's Nest. In *Proceedings of the 2019 International Conference on Management of Data*. 1929–1932.

[21] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland H. C. Yap, Zhenkai Liang, and Manuel Rigger. 2024. Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 46:1–46:12. https://doi.org/10.1145/3597503.3623307

[22] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. (Aug. 2023).

[23] 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

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

[25] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.

[26] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association.

[27] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022,*

Pittsburgh, PA, USA, May 25-27, 2022. ACM, 225–236. https://doi.org/10.1145/3510003.3510093

[28] Michaël Marcozzi, Qiyi Tang, Alastair F Donaldson, and Cristian Cadar. 2019. Compiler fuzzing: How much does it matter? *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

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

[30] 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* (Vancouver, Canada) *(SIGMOD '08)*. ACM, New York, NY, USA, 499–510. https://doi.org/10.1145/1376616.1376668

[31] 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*. 499–510.

[32] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 33–50.

[33] Raghunath Othayoth Nambiar, Meikel Poess, Andrew Masland, H. Reza Taheri, Matthew Emmerton, Forrest Carman, and Michael Majdalany. 2012. TPC Benchmark Roadmap 2012. In *Selected Topics in Performance Evaluation and Benchmarking - 4th TPC Technology Conference, TPCTC 2012, Istanbul, Turkey, August 27, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7755)*, Raghunath Othayoth Nambiar and Meikel Poess (Eds.). Springer, 1–20. https://doi.org/10.1007/978-3-642-36727-4_1

[34] Thomas Neumann. 2009. Query simplification: graceful degradation for join-order optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 403–414. https://doi.org/10.1145/1559845.1559889

[35] Johannes Oetsch, Michael Prischink, Jörg Pührer, Martin Schwengerer, and Hans Tompits. 2012. On the small-scope hypothesis for testing answer-set programs. In *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*.

[36] Krishna Kantikiran Pasupuleti, Jiakun Li, Hong Su, and Mohamed Ziauddin. 2023. Automatic SQL Error Mitigation in Oracle. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3835–3847.

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

[38] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 24–36.

[39] John W Ratcliff, David Metzener, et al. 1988. Pattern matching: The gestalt approach. *Dr. Dobb's Journal* 13, 7 (1988), 46.

[40] 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* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

[41] Manuel Rigger and Zhendong Su. 2020. 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* (Sacramento, California, United States) *(ESEC/FSE 2020)*. https://doi.org/10.1145/3368089.3409710

[42] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (2020). https://doi.org/10.1145/3428279

[43] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Banff, Alberta.

[44] 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* (Boston, Massachusetts) *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

[45] 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* (Portland, OR, USA) *(SIGMOD '20)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC.

[46] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26. https://doi.org/10.1145/3588909

[47] Xiu Tang, Sai Wu, Dongxiang Zhang, Ziyue Wang, Gongsheng Yuan, and Gang Chen. 2023. A Demonstration of DLBD: Database Logic Bug Detection System. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3914–3917.
[48] Website. 1988. TPC-DS Benchmark. https://www.tpc.org/tpcds/. Accessed: 2022-11-15.
[49] Website. 2013. American Fuzzy Lop (AFL) Fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2022-06-08.
[50] Website. 2015. SQLsmith. https://github.com/anse1/sqlsmith. Accessed: 2022-06-08.
[51] Jiaqi Yan, Qiuye Jin, Shrainik Jain, Stratis D Viglas, and Allison Lee. 2018. Snowtrail: Testing with production queries on a cloud database. In *Proceedings of the Workshop on Testing Database Systems*. 1–6.
[52] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970.

Received October 2023; revised January 2024; accepted March 2024