Testing Database Systems via Differential Query Execution

Jiansen Song*†, Wensheng Dou*†‡§, Ziyu Cui*†, Qianwang Dai*†, Wei Wang*†‡§,
Jun Wei*†‡§, Hua Zhong*†, Tao Huang*†
*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
†University of Chinese Academy of Sciences
‡University of Chinese Academy of Sciences Nanjing College
§Nanjing Institute of Software Technology
{songjiansen20, wsdou, cuiziyu20, daiqianwang19, wangwei, wj, zhonghua, tao}@otcaix.iscas.ac.cn

***Abstract*—Database Management Systems (DBMSs) provide efficient data retrieval and manipulation for many applications through Structured Query Language (SQL). Incorrect implementations of DBMSs can result in logic bugs, which cause `SELECT` queries to fetch incorrect results, or `UPDATE` and `DELETE` queries to generate incorrect database states. Existing approaches mainly focus on detecting logic bugs in `SELECT` queries. However, logic bugs in `UPDATE` and `DELETE` queries have not been tackled.**

In this paper, we propose a novel and general approach, which we have termed *Differential Query Execution* (*DQE*), to detect logic bugs in `SELECT`, `UPDATE` and `DELETE` queries of DBMSs. The core idea of DQE is that different SQL queries with the same predicate usually access the same rows in a database. For example, a row updated by an `UPDATE` query with a predicate φ should also be fetched by a `SELECT` query with the same predicate φ. If not, a logic bug is revealed in the target DBMS. To evaluate the effectiveness and generality of DQE, we apply DQE on five production-level DBMSs, i.e., MySQL, MariaDB, TiDB, CockroachDB and SQLite. In total, we have detected 50 unique bugs in these DBMSs, 41 of which have been confirmed, and 11 have been fixed. We expect that the simplicity and generality of DQE can greatly improve the reliability of DBMSs.

***Index Terms*—Database system, DBMS testing, logic bug**

I. Introduction

Database Management Systems (DBMSs) are designed to efficiently retrieve and manipulate data in databases. Relational DBMSs, e.g., MySQL [1], MariaDB [2], TiDB [3], CockroachDB [4] and SQLite [5], adopt Structured Query Language (SQL) [6] as their standard query language, and have become an indispensable component in many business-critical applications [7].

DBMSs suffer from various bugs, e.g., crashes and logic bugs. Specially, logic bugs can cause a DBMS to return incorrect results for `SELECT` queries, or generate incorrect database states for `UPDATE` and `DELETE` queries. Such logic bugs do not crash the DBMS, and can easily go unnoticed by developers. In this work, we focus on detecting logic bugs in DBMSs.

Recently, researchers have proposed some approaches to detect logic bugs in DBMSs [8]–[11]. RAGS [8] feeds a `SELECT` query into multiple DBMSs and observes discrepancies in their query results. PQS [9] generates `SELECT` queries that fetch a pivot row, and checks whether the target DBMS fails to fetch the pivot row. NoREC [10] rewrites a `SELECT` query as another equivalent one that cannot be optimized by the DBMS, and then detects difference in their query results. TLP [11] decomposes a `SELECT` query into three partitioning queries, and merges these partitioning queries' results into a combined result, which is expected to be the same as the original query's result. However, all these approaches mainly focus on detecting logic bugs in `SELECT` queries. While the logic bugs in `UPDATE` and `DELETE` queries have not been tackled yet, even though they can cause severer consequences, e.g., incorrect database states.

Logic bugs in DBMSs, especially those in `UPDATE` and `DELETE` queries, are difficult to detect automatically. A key challenge to detect logic bugs is to construct an effective test oracle to determine whether a DBMS behaves correctly for a given query. Existing approaches to construct oracles for `SELECT` queries, e.g., PQS [9], NoREC [10] and TLP [11], cannot be adopted on `UPDATE` and `DELETE` queries.

In DBMSs, `SELECT`, `UPDATE` and `DELETE` queries utilize predicates (i.e., `WHERE` clauses) to specify which rows to retrieve, update or delete, respectively. If they use the same predicate φ, they should access the same rows in a database. Ideally, DBMSs can adopt the same implementations for predicate evaluation in `SELECT`, `UPDATE` and `DELETE` queries. However, a DBMS usually adopts different implementations for predicate evaluation in `SELECT`, `UPDATE` and `DELETE` queries due to various optimization choices[1]. Inconsistent implementations for predicate evaluation among these queries can cause `SELECT`, `UPDATE` and `DELETE` queries with the same predicate φ to access different rows.

Inspired by this key observation, we propose *Differential Query Execution* (*DQE*), a novel and general approach to detect logic bugs in `SELECT`, `UPDATE` and `DELETE` queries. DQE solves the test oracle problem by executing `SELECT`, `UPDATE` and `DELETE` queries with the same predicate φ, and observing inconsistencies among their execution results. For example, if a row that is updated by an `UPDATE` query with a predicate φ does not appear in the query result of a

Wensheng Dou and Hua Zhong are the corresponding authors.

[1]https://bugs.mysql.com/bug.php?id=106420

SELECT query with the same predicate φ, a logic bug is detected in the target DBMS. The key challenge of DQE is to automatically obtain the accessed rows for a given SELECT, UPDATE or DELETE query. To address this challenge, we append two extra columns to each table in a database, to uniquely identify each row and track whether a row has been modified, respectively. We further rewrite SELECT and UPDATE queries to identify their accessed rows.

```
CREATE TABLE t1 (c1 INT);
INSERT INTO t1 VALUES (1); -- r1
1. SELECT * FROM t1 WHERE '';
    -- Fetch empty result
    -- Warning|1292|Truncated incorrect DOUBLE
        value ''
2. UPDATE t1 SET c1=2 WHERE '';
    -- Update r1
3. DELETE FROM t1 WHERE '';
    -- Delete r1
```

Listing 1. TiDB#27648. The UPDATE and DELETE queries unexpectedly change the database state.

Listing 1, shows a real-world logic bug TiDB#27648[2] detected by DQE. In this bug, the UPDATE and DELETE queries unexpectedly change the database state. Table $t1$ consists of an INT row with value 1. The predicate φ is ' ', i.e., an empty string. TiDB tries to convert φ into a boolean value for the three queries at Line 1−3. For the SELECT query, TiDB first truncates φ to a DOUBLE value 0, and then converts it to a boolean value FALSE. Therefore, the SELECT query fetches an empty query result and raises a warning. For the UPDATE and DELETE queries, TiDB erroneously evaluates φ to TRUE, and changes the database state unexpectedly. We report this bug to TiDB developers, who have confirmed and fixed it. Existing approaches cannot detect this bug, because this bug occurs in the UPDATE and DELETE queries.

To evaluate DQE's effectiveness and generality, we implement DQE and perform experiments on five widely-used and production-level DBMSs, i.e., MySQL [1], MariaDB [2], TiDB [3], CockroachDB [4] and SQLite [5]. In total, we have detected 50 unique bugs among these DBMSs, 41 of which have been confirmed as new bugs, and 11 bugs have been fixed. Among the 41 confirmed bugs, 20 bugs occur in UPDATE and DELETE queries. None of our detected bugs can be detected by existing approaches, e.g., PQS [9], NoREC [10] and TLP [11]. Our experimental results indicate that DQE is effective in detecting logic bugs in SELECT, UPDATE and DELETE queries in DBMSs. We have made DQE publicly available at https://github.com/tcse-iscas/dqetool.

Although we have detected many bugs in SELECT, UPDATE and DELETE queries in our target DBMSs, DQE still has some limitations. First, DQE suffers from the same issue as differential testing, in which DQE fails to detect the same bug occurring in all the three SELECT, UPDATE and DELETE queries. Second, DQE only supports common operations and functions in SELECT, UPDATE and DELETE queries, e.g., JOIN, ORDER BY, and LIMIT. DQE does not support operations and functions that are only used in one kind of queries, e.g., DISTINCT, sub-queries, aggregate-based functions, window functions and GROUP BY that are only used in SELECT queries. For these features, DQE cannot compare their execution results in SELECT, UPDATE and DELETE queries. Third, DQE cannot support non-deterministic functions, e.g., RAND function, which returns different values in different queries.

Despite these limitations, the key insight of DQE is widely applicable to other DBMSs that support data manipulation specified by predicates, e.g., `find()`, `update()` and `remove()` in MongoDB. We expect that DQE can be widely adopted to improve the reliability of DBMSs and draw attention to detecting logic bugs in UPDATE and DELETE queries.

In summary, we make the following contributions.

- We propose DQE, a novel and general approach to detect logic bugs in SELECT, UPDATE and DELETE queries in DBMSs. To our knowledge, DQE is the first approach to detect logic bugs in UPDATE and DELETE queries.
- We implement and evaluate DQE on five widely-used DBMSs. DQE has detected 41 previously-unknown bugs in these DBMSs, 20 of which occur in UPDATE and DELETE queries.

II. Preliminaries

We first explain our target DBMSs (Section II-A and Section II-B), and then discuss SQL query execution strategies that are adopted in our target DBMSs (Section II-C).

A. Database Management Systems and SQL

Database Management Systems (DBMSs) are widely used in many applications for effective data retrieval and manipulation. Mainstream DBMSs, e.g., MySQL [1], MariaDB [2], TiDB [3], CockroachDB [4] and SQLite [5], adopt the *relational data model* [12], which organizes data into relational tables. These DBMSs are so-called relational DBMSs.

Relational DBMSs usually adopt *Structured Query Language (SQL)* [6] as their query language. In SQL, SELECT, UPDATE and DELETE queries utilize predicates (i.e., WHERE clauses) to determine which rows to retrieve, update or delete, respectively. DBMSs usually adopt sophisticated optimizations to increase the performance of query evaluation [13]–[16]. For the same predicate, DBMSs can apply different optimizations in SELECT, UPDATE and DELETE queries. For example, MySQL developers stated that *"all the DML statements have to pass through the optimizing stage,... SELECT and UPDATE queries do not pass through the same optimizing process."*[3] However, no matter what optimizations are applied on query evaluation, SELECT, UPDATE and DELETE queries with the same predicate φ should access the same rows.

B. Target DBMSs

We focus on five production-level and widely-used DBMSs, i.e., MySQL [1], MariaDB [2], TiDB [3], CockroachDB [4]

[2]https://github.com/pingcap/tidb/issues/27648

[3]https://bugs.mysql.com/bug.php?id=106420

TABLE I
TARGET DBMSS

DBMS	DB-Engines Ranking	GitHub Stars	Type
MySQL	2	8.7K	Traditional
MariaDB	13	4.7K	Traditional
TiDB	108	33.3K	NewSQL
CockroachDB	57	26.5K	NewSQL
SQLite	9	3.5K	Embedded

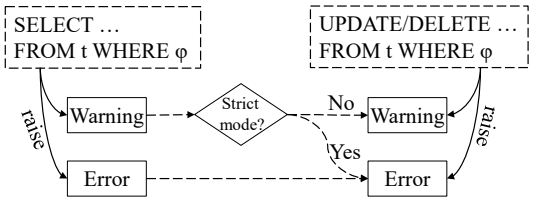

Fig. 1. SQL modes in MySQL, MariaDB and TiDB.

and SQLite [5], as shown in Table I. We choose these DBMSs based on their popularity and database types. The DB-Engines Ranking [17] shows that MySQL, SQLite and MariaDB are among the most popular DBMSs, which are ranked 2nd, 9th and 13th, respectively. MySQL and MariaDB are traditional DBMSs that have been developed for decades. SQLite is an embedded DBMS and the most widely deployed DBMS [18]. According to GitHub Database Topic [19], TiDB and CockroachDB are the top two popular (33.3K and 26.5K stars, respectively) relational DBMSs. CockroachDB and TiDB are distributed NewSQL DBMSs with high scalability.

C. Query Execution Strategy

Different DBMSs usually adopt different SQL query execution strategies, e.g., how to handle syntax and semantic errors in query evaluation. In this section, we mainly discuss about how our target DBMSs handle syntax and semantic errors in query evaluation for `SELECT`, `UPDATE` and `DELETE` queries, which can affect the query execution analysis in DQE.

MySQL, MariaDB and TiDB. These three DBMSs adopt the same query execution strategies. When a syntax or semantic error err occurs in query evaluation, DBMSs can raise warnings or errors according to the severity of err. For example, if an invalid value (e.g., comparing an INTEGER value with a TEXT value) is used, a warning is raised, while if a predicate is syntactically invalid (e.g., a function takes more arguments than necessary), an error is raised. If a warning occurs when evaluating a query, DBMSs can continue to execute the query. For example, when evaluating a predicate φ on row $r1$ raises a warning, DBMSs can continue to evaluate φ on the following rows, e.g., row $r1+1$. If an error occurs when evaluating a query, DBMSs will abort and roll back the query. Specifically, `SELECT` queries return an empty query result, and all changes made by `UPDATE` and `DELETE` queries are rolled back.

These three DBMSs can execute queries in different SQL modes, which can affect the query execution strategies. A SQL mode is a set of configurations, e.g., `STRICT_ALL_TABLES` and `STRICT_TRANS_TABLES`. Specially, there are two SQL modes, i.e., strict mode and non-strict mode, which can affect query execution strategies of `SELECT`, `UPDATE` and `DELETE` queries. Fig. 1 shows how SQL queries handle warnings and errors for a given predicate φ in different SQL modes. When enabling strict mode, DBMSs adopt strict validation check for `UPDATE` and `DELETE` queries. Specifically, if a `SELECT` query with a predicate φ raises a warning, the warning is treated as an error (with the same error message) in the `UPDATE` and `DELETE` queries with the same predicate φ. For non-strict mode, if a `SELECT` query with a predicate φ raises a warning, the `UPDATE` and `DELETE` queries with the same predicate φ raise the same warning, too. Note that, if a `SELECT` query with a predicate φ raises an error, the `UPDATE` and `DELETE` queries with the same predicate φ also raise the same error no matter whether strict mode is enabled.

CockroachDB and SQLite. These two DBMSs adopt relatively simple query execution strategies. In these two DBMSs, `SELECT`, `UPDATE` and `DELETE` queries can only raise errors when a syntax or semantic error occurs in predicate evaluation, and do not raise warnings. If an `UPDATE` or `DELETE` query raises an error, it will be rolled back, and all changes made by the query will be undone. However, if a `SELECT` query raises an error, it will return all rows that match its predicate before the error occurs. That said, a `SELECT` query may return a non-empty query result when it raises an error.

III. APPROACH

We propose *Differential Query Execution* (*DQE*) to automatically detect logic bugs in `SELECT`, `UPDATE` and `DELETE` queries. The core idea of DQE is that the `SELECT`, `UPDATE` and `DELETE` queries with the same predicate φ should access the same rows. If these queries access different rows, DQE reveals a potential logic bug in the target DBMS.

A. DQE Overview

Fig. 2 shows the workflow of DQE. We first generate a random database (step ①). The generated database contains one or more tables, e.g., $t1$ and $t2$. Each table contains some random columns and data, e.g., table $t1$ has a column $c1$ with value 'a' and 'b'. We then randomly generate a predicate φ, e.g., $NOT\ c1$ (step ②). Based on predicate φ, we generate a query triple $< Q_{sel}, Q_{up}, Q_{del} >$, in which Q_{sel}, Q_{up} and Q_{del} denote a `SELECT` query, an `UPDATE` query and a `DELETE` query, respectively. Q_{sel}, Q_{up}, and Q_{del} in the query triple all use φ as their predicates (step ③). We then execute Q_{sel}, Q_{up} and Q_{del} in the query triple on the same database state (step ④), and analyze their execution results, i.e., accessed rows and raised errors (step ⑤). Specially, we analyze Q_{sel}'s query result rs, the modified tables t_u and t_d after executing Q_{up} and Q_{del}, respectively. If the three queries' execution results in the query triple are inconsistent, e.g., accessing different rows, DQE reveals a potential logic bug in the target DBMS (step ⑥).

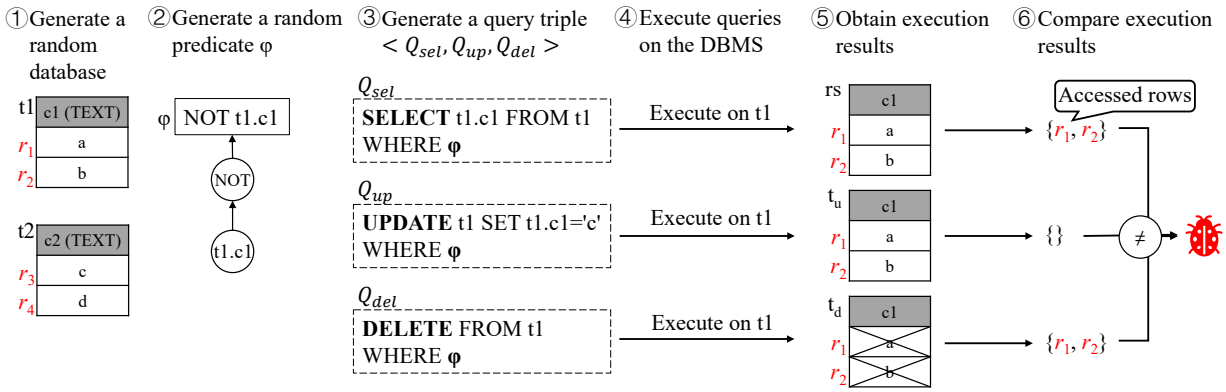

Fig. 2. We use MariaDB#27885 [20] to illustrate the workflow of DQE. rs denotes the query result of Q_{sel}, t_u denotes $t1$'s new table state after executing Q_{up}, and t_d denotes $t1$'s new table state after executing Q_{del}.

The rest of this section is organized as follows. Section III-B describes our database generation. Section III-C describes our strategies to generate SQL queries. Section III-D shows how we obtain a query triple's execution results. Section III-E shows how we detect logic bugs by comparing the execution results of the three queries in a query triple.

B. Database Generation

Database generation has been widely explored by existing works [21]–[26], and is not a contribution of this work. Our database generation is mainly adopted from SQLancer [27]. We present our database generation only for completeness.

We first use the `CREATE TABLE` command to create at most $maxTable$ tables. Each table contains at most $maxCol$ columns. We assign each column with a random column type, e.g., `INT` or `TEXT`, and some column constraints, e.g., `PRIMARY KEY` and `UNIQUE`. We then populate random data into each table by executing the `INSERT` command. Each table contains at most $maxInsert$ rows of data. We further execute at most $maxAlter$ `ALTER TABLE` and `CREATE INDEX` commands to modify each initial table, e.g., adding new columns or building indexes on existing columns. Moreover, we configure each table with random options, e.g., setting the starting number of the auto-incrementing column by appending `AUTO_INCREMENT=5`. Note that, $maxTable$, $maxCol$, $maxInsert$, $maxAlter$ are all configurable parameters. We set them as 5, 3, 10, 3 by default in our experiment, respectively.

After generating the initial database, we alter each table in the database by adding column $rowId$ and $updated$. Column $rowId$ is used to uniquely identify each row. We assign column $rowId$ as `TEXT` type and populate it with unique values, e.g., UUID. Column $updated$ is used to track the modifications of each row. We assign column $updated$ as `INT` type with default value 0. Note that, these two newly-added columns are not used in the following query triple generation (Section III-C). We only use them to obtain a query's accessed rows in Section III-D.

The above database generation is specific to individual DBMSs. Different DBMSs support different column types, column constraints and table options. For example, CockroachDB supports column type `INTERVAL`, while MySQL does not support it.

Algorithm 1: Predicate generation.

```
   Input: maxDepth
1  Function generatePredicate () do
2      generateAST(0)
3  Function generateAST(depth) do
4      nodeTypes ← {CONST, COLUMN}
5      if depth < maxDepth then
6          nodeTypes ←
             {CONST, COLUMN, AND, OR, ...}
7      nodeType ← random(nodeTypes)
8      if nodeType = CONST then
9          return randomConst()
10     else if nodeType = COLUMN then
11         return randomTable().randomColumn()
12     else
13         node ← nodeType.createNode()
14         for i ← 1; i ≤ node.operands.length; i++ do
15             node.child[i] ← generateAST
                 (depth + 1)
16         return node
```

C. Query Triple Generation

After database generation, we generate a query triple $< Q_{sel}, Q_{up}, Q_{del} >$, in which Q_{sel}, Q_{up}, Q_{del} use the same predicate φ. In the following paragraphs, we first explain how to generate a predicate φ, and then explain how to generate Q_{sel}, Q_{up}, and Q_{del} based on predicate φ.

Predicate generation. We use Algorithm 1 to randomly generate predicates based on Abstract Syntax Trees (ASTs) of SQL. We randomly choose one node type from $CONST$, $COLUMN$, and operators supported by the target DBMS (Line 5−7). If the node type is $CONST$, we randomly

TABLE II
EXAMPLES OF UPDATE OR DELETE -SPECIFIC ERRORS IN MYSQL AND SQLITE

Table Constraint	MySQL	SQLite
NOT NULL	1048, Column 'c0' cannot be null	NOT NULL constraint failed: t1.c1
UNIQUE	1062, Duplicate entry '2' for key 't1.i0'	UNIQUE constraint failed: t1.c1
Generated Column	3105, The value specified for generated column 'c1' in table 't1' is not allowed	cannot UPDATE generated column "c2"
Foreign Key	1451, Cannot delete or update a parent row: a foreign key constraint fails	FOREIGN KEY constraint failed

generate a constant value, e.g., 'a' (Line 8−9). If the node type is $COLUMN$, we randomly return a column reference from the tables in our generated database, e.g., $t1.c1$ (Line 10−11). If the node type is an operator, we iteratively generate its operands (Line 13−15). When the depth of an AST reaches $maxDepth$, we only generate a constant or a column reference (Line 4−6) and will not expand the AST further. Note that, $maxDepth$ is configurable, and we set it as 3 by default in our experiment.

Query triple generation. After generate a predicate φ, we further randomly generate Q_{sel}, Q_{up}, and Q_{del} based on predicate φ. Specially, We first extract the referenced tables from predicate φ, which are used in Q_{sel}, Q_{up}, and Q_{del}. For example, if predicate φ is $t1.c1 > 2\ AND\ t2.c2 > 1$, its referenced tables are $t1$ and $t2$. We then generate Q_{sel}'s select field and Q_{up}'s update field. The select field of Q_{sel} is a list of column references, e.g., $t1.c1, t2.c2$. The update field of Q_{up} is a list of assignments, e.g., $t1.c1 = 1, t2.c2 = 2$. Finally, we generate optional clauses, e.g., `ORDER BY`, which are commonly supported by Q_{sel}, Q_{up}, and Q_{del}. For example, we can generate a query triple as follows.

```
Q_sel: SELECT t1.c1,t2.c2 FROM t1,t2
           WHERE t1.c1>2 AND t2.c2>1
Q_up:  UPDATE t1,t2 SET t1.c1=1,t2.c2=2
           WHERE t1.c1>2 AND t2.c2>1
Q_del: DELETE t1,t2 FROM t1,t2
           WHERE t1.c1>2 AND t2.c2>1
```

During query generation, DQE supports common operations and functions in `SELECT`, `UPDATE` and `DELETE` queries, and does not support operations and functions that are only used in one type of queries, e.g., `DISTINCT`, aggregate-based functions, window functions and `GROUP BY` that are only used in `SELECT` queries. Moreover, DQE cannot support non-deterministic functions, e.g., `RAND` function that returns a random value. Note that, our query generation is specific to DBMSs.

D. Obtaining Execution Results

For a generated query triple $< Q_{sel}, Q_{up}, Q_{del} >$, we execute Q_{sel}, Q_{up} and Q_{del} on the same database state, and obtain their execution results. Note that, the three queries' execution results, i.e., Q_{sel}'s query result, the modified tables by Q_{up}, and Q_{del}, cannot be used directly to compare and find bugs. Instead, we collect two kinds of information in each query, i.e., the rows accessed by a query, and the errors raised by a query if any. These information can be used to compare the queries' execution results in the query triple in Section III-E. In the following, we first discuss how to obtain the errors raised by a query, and then discuss how to automatically obtain each query's accessed rows in a query triple in details.

1) Obtaining the errors raised by a query: For a query, e.g., Q_{sel}, Q_{up} and Q_{del}, it can raise errors (sometimes warnings in MySQL, MariaDB and TiDB) when a syntax or semantic error occurs in query evaluation. Specially, an `UPDATE` query Q_{up} may violate table constraints, e.g., `NOT NULL` and `UNIQUE`, and raises `UPDATE`-specific errors when updating the referenced tables. Similarly, a `DELETE` query Q_{del} may violate table constraints, e.g., `FOREIGN KEY`, and raises `DELETE`-specific errors when deleting data in the referenced tables. Table II shows some errors about constraint violations in MySQL and SQLite for `UPDATE` and `DELETE` queries. For example, when Q_{up} updates on a column $c1$ with `NOT NULL` constraint by changing its value to $NULL$, MySQL raises an error with code 1048 and message "Column 'c0' cannot be null". When Q_{del} deletes a column that is referenced by another table, SQLite raises an error with message "FOREIGN KEY constraint failed". Note that, Q_{sel} does not raise specific errors that Q_{up} and Q_{del} cannot raise.

We use diagnostic commands provided by our target DBMSs to obtain the errors raised by a query. Specially, we use the `SHOW WARNINGS` command to obtain the raised errors in MySQL, MariaDB and TiDB. The `SHOW WARNINGS` command returns the error level, code and message when executing a query [28]. CockroachDB and SQLite do not provide such diagnostic commands. Thus, we use `SQLException` in Java to obtain the raised errors in these two DBMSs.

2) Obtaining the accessed rows by a `SELECT` query (Q_{sel}): In order to obtain the rows returned by a `SELECT` query Q_{sel}, we append the select field of Q_{sel} with column $rowId$ of its referenced tables, and form a new `SELECT` query Q'_{sel}. After executing Q'_{sel}, we fetch column $rowId$'s values from its result set. Because column $rowId$'s values can appear more than once when testing multiple tables, we remove the duplicate values of column $rowId$ if necessary.

Fig. 3 shows an example for Q_{sel}. We append Q_{sel}'s select field with $t1.rowId, t2.rowId$, which is shown in the red font, and form Q'_{sel}. We then execute Q'_{sel} to get the accessed rows from its query result rs. In this example, the same value of column $t1.rowId$ appears twice, so we remove the duplicate values. We can see that the accessed rows by Q_{sel} is $r3, r5, r6$.

3) Obtaining the accessed rows by an `UPDATE` query (Q_{up}): In order to obtain the rows updated by an `UPDATE` query Q_{up}, we append the update field of Q_{up} with a list of assignments to column $updated$ in its referenced tables, and

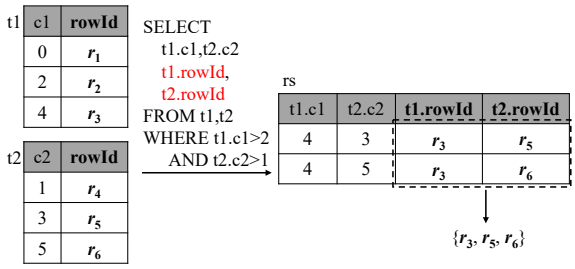

Fig. 3. Obtaining the accessed rows by a SELECT query.

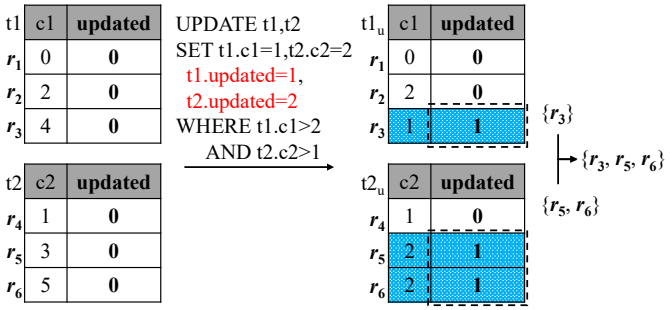

Fig. 4. Obtaining the accessed rows by an UPDATE query.

form a new UPDATE query Q'_{up}. After executing Q'_{up}, we fetch column $rowId$'s values from each referenced table with column $updated$'s value equal to 1, and form the accessed rows by Q_{up}.

Fig. 4 shows an example for Q_{up}. We append Q_{up}'s update field with $t1.updated = 1, t2.updated = 1$, which is shown in the red font, and form Q'_{up}. We then execute Q'_{up}, and obtain its accessed rows by combining column $rowId$ in each modified table (i.e., $t1_u$ and $t2_u$) with column $updated$'s value equal to 1. We can see that the accessed rows by Q_{up} is $r3, r5, r6$.

4) Obtaining the accessed rows by a DELETE query (Q_{del}): In order to obtain the rows deleted by a DELETE query Q_{del}, we compare column $rowId$'s values in each referenced table before and after executing it.

Fig. 5 shows an example for Q_{del}. Before executing Q_{del}, the column $rowId$'s values in table $t1$ are $r1, r2, r3$, and the column $rowId$'s values in table $t2$ are $r4, r5, r6$. After executing Q_{del}, the column $rowId$'s values in the modified table $t1_d$ are $r1, r2$, and the column $rowId$'s values in the modified table $t2_d$ are $r4$, which means $r3$ in the table $t1$ and $r5, r6$ in the table $t2$ are deleted. Therefore, the accessed rows by Q_{del} is $r3, r5, r6$.

Note that, in Fig. 3, Fig. 4, and Fig. 5, we only show the related columns for brevity. In fact, we append these two columns to each table in our database generation.

E. Comparing Execution Results

After obtaining the execution results of Q_{sel}, Q_{up} and Q_{del}, we analyze and compare them to detect whether a logic bug occurs in the target DBMS. In the following, we use row_{sel}, row_{up} and row_{del} to present the set of accessed rows by Q_{sel}, Q_{up} and Q_{del}, respectively.

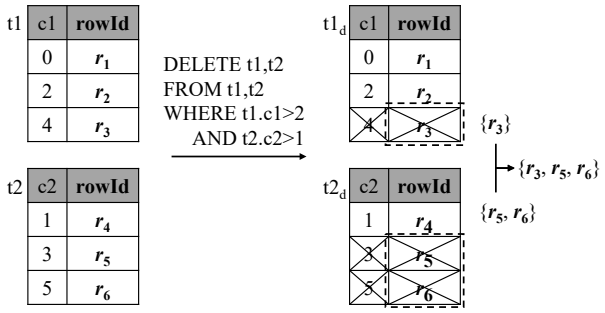

Fig. 5. Obtaining the accessed rows by a DELETE query.

As discussed in Section III-D, Q_{up} and Q_{del} can raise UPDATE and DELETE -specific errors, respectively, while Q_{sel} cannot. Therefore, if Q_{up} raises UPDATE-specific errors, we will not compare Q_{up}'s execution results with those of Q_{sel} and Q_{del}, i.e., we only compare the execution results of Q_{sel} and Q_{del}. Similarly, if Q_{del} raises DELETE-specific errors, we will not compare Q_{del}'s execution results with those of Q_{sel} and Q_{up}. In these two cases, row_{up} or row_{del} should be empty. In the following discussion, we assume that Q_{up} and Q_{del} do not raise UPDATE and DELETE -specific errors, respectively.

As discussed in Section II-C, different DBMSs adopt different query execution strategies. Thus, we first discuss how to compare a query triple's execution results in MySQL, MariaDB and TiDB, and then discuss how to compare a query triple's execution results in CockroachDB and SQLite.

1) MySQL, MariaDB and TiDB: We apply the following rules to compare the execution results of Q_{sel}, Q_{up} and Q_{del} in a query triple. If any rule is violated, DQE reports a bug.

- If Q_{sel} raises an error, Q_{up} and Q_{del} should raise the same error. In this case, row_{sel}, row_{up} and row_{del} should be empty.
- Under strict mode, if Q_{sel} raises a warning, Q_{up} and Q_{del} should raise an error. The warning and error should have the same error codes and messages, except for their error levels. In this case, row_{up} and row_{del} should be empty.
- Under non-strict mode, if Q_{sel} raises a warning, Q_{up} and Q_{del} should raise the same warning. In this case, row_{sel}, row_{up} and row_{del} should be the same.
- If Q_{sel} does not raise a warning or an error, Q_{up} and Q_{del} should not raise a warning or an error. In this case, row_{sel}, row_{up} and row_{del} should be the same.

2) CockroachDB and SQLite: We apply the following rules to compare the execution results of Q_{sel}, Q_{up} and Q_{del} in a query triple. If any rule is violated, DQE reports a bug.

- If Q_{sel} raises an error, Q_{up} and Q_{del} should raise the same error. In this case, row_{up} and row_{del} should be empty. However, row_{sel} may not be empty, as discussed in Section II-C.

TABLE III
BUGS REPORTED BY DQE

DBMS	Bug Status					Triggering Query		
	Submitted	Confirmed	Fixed	Duplicate	Not a bug	SELECT	UPDATE	DELETE
MySQL	7	1	1	0	6	0	1	1
MariaDB	4	2	0	0	0	0	0	2
TiDB	37	37	10	0	0	20	17	17
CockroachDB	1	0	0	1	0	0	0	0
SQLite	1	1	0	0	0	1	0	0
Total	**50**	**41**	**11**	**1**	**6**	**21**	**18**	**20**

- If Q_{sel} does not raise a warning or an error, Q_{up} and Q_{del} should not raise a warning or an error. In this case, row_{sel}, row_{up} and row_{del} should be the same.

IV. EVALUATION

We implement DQE based on SQLancer [27], which is implemented in Java. We make the following improvements to apply DQE. First, we add `UPDATE` and `DELETE` query generation in our target DBMSs, e.g., MySQL, TiDB and SQLite. Second, DQE requires to execute the three queries in a query triple on the same database state. In MySQL, MariaDB and TiDB, we use `ROLLBACK` transactions to roll back all changes made by `UPDATE` and `DELETE` queries. In CockroachDB and SQLite, we record the table content before query execution and refill the table with the same content after query execution. In total, we write about 2,600 lines of code to implement DQE on five target DBMSs.

We evaluate the effectiveness of DQE by answering the following two research questions:

- **RQ1:** *What logic bugs can DQE detect in real-world DBMSs?*
- **RQ2:** *How many bugs detected by DQE can be found by existing approaches?*

A. Experimental Methodology

Experimental setup. We evaluate DQE on five widely-used DBMSs, i.e., MySQL, MariaDB, TiDB, CockroachDB and SQLite. Detailed information about these DBMSs are presented in Section II-B. We test these DBMSs with their latest release versions when we start our experiment, i.e., MySQL 8.0.28, MariaDB 10.8.2, TiDB 5.2.0, CockroachDB 21.2.6 and SQLite 3.39.2. For TiDB, we also test version 5.3.0 and 5.4.0 after they are released.

We perform our experiment on a CentOS machine with 8 CPU cores and 32GB RAM. We deploy our target DBMSs according to their own deployment requirements. Specifically, we deploy MySQL and MariaDB using Docker containers. We deploy TiDB in a local cluster with a TiKV instance, a TiDB instance and a PD instance. We deploy CockroachDB in a local cluster with three nodes. We embed SQLite within DQE.

Experimental process. We run DQE to find bugs in our target DBMSs. We do not set timeout for our experiment and continuously run DQE until it finds bugs. The whole experiment takes about one month. When DQE reports a potential bug, we manually execute it through the interactive terminal of the target DBMS to check whether this bug can be reproduced. When successfully reproducing the reported bug, we manually reduce the test case to a smaller size. We apply the following three strategies to perform the test case reduction. First, we remove the unused columns and optional column constraints. Second, we remove data in tables by eliminating all `INSERT` commands that do not change the bug consequence. Third, we randomly remove some sub-clauses in predicates without changing the bug consequence. We then check whether this bug has been reported in the target DBMS's bug tracking system to avoid submitting duplicate bugs. After reporting a bug, we wait for feedbacks from developers.

Experimental focus. The developers' response time determines how much effort is spent on testing a DBMS. TiDB developers give us more responsive confirmation than other DBMS developers, which highly increases our confidence to continue our test. Therefore, we spend most of our testing time on TiDB and keep it up-to-date.

B. Overall Detection Results

To answer RQ1, we evaluate DQE on MySQL, MariaDB, TiDB, CockroachDB and SQLite. In total, DQE reports 122 bugs among them. We manually reproduce and minimize the test cases of these 122 reported bugs. If the minimized test cases of some bugs are the same, we only keep one, and consider others as duplicate bugs. Finally, we obtain 50 unique bugs, and submit them to corresponding DBMS developers. Specifically, we submit 7 bugs in MySQL, 4 bugs in MariaDB, 37 bugs in TiDB, 1 bug in CockroachDB and 1 bug in SQLite. Note that, we do not submit the remaining 72 bugs, which are considered as duplicate bugs by us and not false positives.

Table III shows the bug status of our submitted bugs (column 2-6). Among the 50 submitted bugs, 41 bugs have been confirmed as new bugs, in which 11 bugs have been fixed. Among the 41 confirmed bugs, MySQL developers confirm 1 bug, MariaDB developers confirm 2 bugs, TiDB developers confirm 37 bugs and SQLite developers confirm 1 bug. Among the 11 fixed bugs, MySQL developers fix 1 bug and TiDB developers fix 10 bugs. For the 9 bugs that have not been confirmed, 1 bug in CockroachDB is considered as duplicate to an existing bug, 6 bugs are considered as intended behaviors by MySQL developers, and the remaining 2 bugs have not been decided by MariaDB developers yet.

TABLE IV
BUG CONSEQUENCES IN SELECT, UPDATE AND DELETE QUERIES

Consequence	SELECT	UPDATE	DELETE
Incorrect database state	0	18	18
Duplicate warning	5	0	0
Unexpected warning	6	0	2
Unexpected error	1	0	0
Incorrect warning message	6	0	0
Others	3	0	0
Total	**21**	**18**	**20**

Among the 41 confirmed bugs, 22 bugs are verified as *Major* or *Moderate*. Note that, different DBMSs have different severity levels. We use *Major* to denote *Critical* and *Serious* in MySQL and MariaDB, and *Major* in TiDB. We use *Moderate* to denote *Moderate* in TiDB. Moreover, SQLite developers do not assign a severity level on the confirmed bug, so we do not count it as *Major* or *Moderate*.

Table III also shows the triggering queries of the 41 confirmed bugs (column 7-9). 21 bugs are triggered by `SELECT` queries, 18 bugs are triggered by `UPDATE` queries and 20 bugs are triggered by `DELETE` queries. Note that, one bug can be triggered by more than one type of queries, so the total number of triggering queries is more than the total number of confirmed bugs.

Table IV shows the 41 confirmed bugs' bug consequences with the number of their triggering queries. Most of `SELECT` queries cause incorrect warnings, e.g., duplicate warnings, unexpected warnings and incorrect warning messages. The remaining 3 `SELECT` queries cause the `SHOW WARNINGS` command to fail to return errors. All `UPDATE` queries and most of `DELETE` queries cause incorrect database states. The remaining 2 `DELETE` queries cause unexpected warnings.

Among the 36 queries that lead to incorrect database states, 34 queries occur in TiDB and 2 queries occur in MySQL. All 5 queries that lead to duplicate warnings occur in TiDB. Among the 8 queries that lead to unexpected warnings, 6 queries occur in TiDB and 2 queries occur in MariaDB. One query that leads to unexpected errors occurs in SQLite. All 6 queries that lead to incorrect warning messages occur in TiDB. The remaining 3 queries that lead to the execution failures of the `SHOW WARNINGS` command occur in TiDB.

C. Comparing with Existing Approaches

To answer RQ2, we perform a qualitative comparison with existing approaches (i.e., PQS [9], NoREC [10] and TLP [11]) that aim to detect logic bugs in DBMSs. These three approaches construct oracles to detect logic bugs in single `SELECT` queries. Thus, they cannot detect the 20 logic bugs in `UPDATE` and `DELETE` queries. Moreover, these three approaches do not consider the normal errors that can be unexpectedly raised by `SELECT` queries as logic bugs, e.g., the warnings in Listing 3. Unlike crashes caught by these approaches, these normal errors do not crash the DBMS and need a test oracle to validate their correctness. Thus, they cannot detect 18 logic bugs related to this kind of errors in `SELECT` queries. We further analyze the triggering test cases and bug consequences of the remaining 3 logic bugs in `SELECT` queries. We find that, none of these 3 bugs can be triggered or captured by the oracles in these approaches. Therefore, all our reported bugs cannot be detected by these approaches theoretically.

TABLE V
COVERAGE INFORMATION

Tool	MySQL	MariaDB
PQS	19	-
NoREC	-	18
TLP	18	-
DQE	15	21

Other DBMS testing approaches, e.g., SQLsmith [29], APOLLO [30], AMOEBA [31], RAGS [8] and SparkFuzz [32], cannot construct oracles to detect logic bugs, or cannot detect logic bugs in a single DBMS because differential testing needs multiple DBMSs. Therefore, we do not compare DQE with these approaches.

D. Other Experimental Statistics

Test efficiency. During testing, before a bug we detect is fixed by the DBMS developers, DQE will generate many test cases that trigger the same bug. In total, DQE reports 122 bugs. After filtering out duplicate bugs, we obtain 50 unique bugs. The duplicate rate is 41% (50/122). Existing works [9]–[11] also face the same problem. There is currently no practical way to automatically filter out duplicate test cases for DBMSs. For discovering these 50 unique bugs, we generate 1,776,124,512 query triples.

Query generation efficiency. We measure the query generation efficiency in DQE during testing. In this experiment, we count every queries generated including those that create the database and query triples. In DQE, we generate syntactically valid queries based on Abstract Syntax Trees (ASTs) of SQL. However, SQL in different DBMSs should obey many semantic constraints, which can cause DQE to generate semantically invalid queries. For example, DQE may generate an `INSERT` command that inserts a duplicate value into a `UNIQUE` column. Such semantic errors can lower our success rate of query generation. In MySQL, DQE generates 2,885 queries per second with a success rate of 88%. In MariaDB, DQE generates 3,344 queries per second with a success rate of 87%. In TiDB, DQE generates 1,566 queries per second with a success rate of 89%. In CockroachDB, DQE generates 243 queries per second with a success rate of 72%. In SQLite, DQE generates 12,313 queries per second with a success rate of 97%.

Coverage. To demonstrate the sufficiency of our testing, we compare code coverage with existing works, i.e., PQS [9], NoREC [10] and TLP [11]. We run each tool with the same experimental setting for 24 hours on MySQL and MariaDB[4].

[4]We have not found a suitable way to perform code coverage measurements in TiDB, SQLite and CockroachDB.

Table V shows our experiment results. PQS achieves 19% line coverage in MySQL. NoREC achieves 18% line coverage in MariaDB. TLP achieves 18% line coverage in MySQL. DQE achieves 15% line coverage in MySQL and 21% line coverage in MariaDB. We can see that DQE obtains similar coverage with other works. This is reasonable, since DQE, PQS, NoREC and TLP are all built on SQLancer, which share the similar query generation. Note that, in SQLancer, NoREC does not support testing MySQL, PQS and TLP do not support testing MariaDB. Thus, we do not measure their code coverage.

Although DQE can generate thousands of queries per second in MySQL and MariaDB, the coverage is low. This is expected, because DQE only focuses on query processing in DBMSs. DBMSs also provide many features that we do not test, e.g., user management, configuration, and fault tolerance.

Parameter selection. We use some default parameters, e.g., $maxTable$=5, $maxCol$=10, and $maxDept$=3, to generate databases and queries. These parameters may affect our bug detection effectiveness. However, the impact of these parameters could be low. The reasons are as follows. (1) Logic bugs in DBMSs usually obey the small scope hypothesis [33]. That said, a high proportion of logic bugs can be found by test inputs within some small scopes, e.g., a small number of tables and rows. (2) After minimizing our 50 submitted bugs, we find that all 50 bugs can be detected on a single table, 47 bugs can be detected with one row. $maxTable$, $maxCol$ and $maxDept$ for these 50 submitted bugs are 1, 2 and 3, respectively.

E. Selected Bugs

In this section, we present some interesting bugs detected by DQE according to their bug consequences. Table IV shows the overall statistics of bug consequences. In the following discussion, we illustrate each bug consequence using a representative bug.

```
CREATE TABLE t1 (c1 INT);
INSERT INTO t1 VALUES (1); -- r1
1. SELECT * FROM t1 WHERE 0 ^ '0.5';
    -- Fetch empty result
    -- Warning|1292|Truncated incorrect INTEGER
       value: '0.5'
2. UPDATE t1 SET c1 = 2 WHERE 0 ^ '0.5';
    -- Update r1
3. DELETE FROM t1 WHERE 0 ^ '0.5';
    -- Delete r1
```

Listing 2. TiDB#31708. The UPDATE and DELETE queries unexpectedly change the database state.

Incorrect database state. Listing 2 shows a bug TiDB#31708[5], in which the `UPDATE` and `DELETE` queries unexpectedly change the database states. Table $t1$ consists of an `INT` row with value 1. The predicate φ is 0 ^ '0.5'. For the `SELECT` query, TiDB first convert '0.5' into an `INT` value 0 and then calculates 0 ^ 0 that is equal to 0 (`FALSE`), and finally returns an empty query result. For the `UPDATE` and `DELETE` queries, TiDB unexpectedly evaluates the predicate to `TRUE` and changes the database state without raising any errors.

[5]https://github.com/pingcap/tidb/issues/31708

```
CREATE TABLE t1 (c1 FLOAT);
INSERT INTO t1 VALUES (0); -- r1
1. SELECT c1 FROM t1 WHERE c1 = 'a';
    -- Fetch r1
    -- Warning|1292|Truncated incorrect DOUBLE
       value: 'a'
    -- Warning|1292|Truncated incorrect DOUBLE
       value: 'a'
    -- Warning|1292|Truncated incorrect DOUBLE
       value: 'a'
2. UPDATE t1 SET c1 = 1 WHERE c1 = 'a';
    -- Update no row
    -- Error|1292|Truncated incorrect INTEGER
       value: 'a'
3. DELETE FROM t1 WHERE c1 = 'a';
    -- Delete no row
    -- Error|1292|Truncated incorrect INTEGER
       value: 'a'
```

Listing 3. TiDB#31711. The SELECT query raises duplicate warnings.

Duplicate warning. Listing 3 shows a bug TiDB#31711[6], in which the `SELECT` query raises three same warnings on row $r1$. Table $t1$ consists of a `FLOAT` row with value 0. The predicate φ is c1 = 'a'. TiDB evaluates it by checking whether column $c1$'s value is equal to constant 'a'. Because there is only one row $r1$ in table $t1$, TiDB should evaluate the predicate φ only once. Note that, column $c1$ is in different data type with constant 'a', which is a string. Therefore, TiDB requires a type conversion, i.e., converting 'a' to a `DOUBLE` value 0, when evaluating the predicate φ. For the `SELECT` query, TiDB returns three warnings to indicate such conversions. These warnings will confuse users, because these three same warning are raised on the same row $r1$. Note that, the warning message raised by the `UPDATE` and `DELETE` queries is also incorrect, because TiDB should convert the constant 'a' to a `DOUBLE` value according to its reference manual [34], instead of `INT` type.

```
CREATE TABLE t1 (c1 BLOB);
INSERT INTO t1 VALUES ('a'); -- r1
1. SELECT * FROM t1 WHERE c1;
-- Fetch empty result
-- Warning|1292|Truncated incorrect DOUBLE value
    : 'a'
2. UPDATE t1 SET c1 = 'b' WHERE c1;
-- Update no row
-- Error|1292|Truncated incorrect DOUBLE value:
    'a'
3. DELETE FROM t1 WHERE c1;
-- Delete no row
-- Warning|1292|Truncated incorrect DOUBLE value
    : 'a'
```

Listing 4. MariaDB#28140. The DELETE query raises an unexpected warning.

Unexpected warning. Listing 4 shows a bug MariaDB#28140[7] in the strict mode, in which the `DELETE` query raises a warning instead of an error. In this bug, because the `SELECT` query raises a warning, the same warning should be treated as an error in the `DELETE` query. However, the `DELETE` query raises a warning. In this bug, because the predicate φ is $c1$, which value is 'a', the `DELETE` query

[6]https://github.com/pingcap/tidb/issues/31711
[7]https://jira.mariadb.org/browse/MDEV-28140

evaluates φ to FALSE. Therefore, the DELETE query does not change the database state. However, such unexpected warning can lead to change the database state in some cases. For example, if predicate φ is NOT $c1$, the DELETE query will change the database state to an empty table unexpectedly.

```
CREATE TABLE t1 (c1 TEXT);
INSERT INTO t1 VALUES ('a'); -- r1
1. SELECT c1 FROM t1 WHERE (NULL == c1) AND
    json_object(c1, c1);
    -- Fetch no row
    -- Runtime error: json_object() labels must
        be TEXT
2. UPDATE t1 SET c1 = 'b' WHERE (NULL == c1) AND
     json_object(c1, c1);
    -- Update no row
3. DELETE FROM t1 WHERE (NULL == c1) AND
    json_object(c1, c1);
    -- Delete no row
```

Listing 5. SQLite#12638. The SELECT query raises an unexpected error.

Unexpected error. Listing 5 shows a bug SQLite#12638 [8], in which the SELECT query should not raise an error. Table $t1$ consists of a TEXT row with value 'a'. The predicate φ is (NULL == c1) AND json_object(c1, c1). The json_object function accepts a pair of arguments, e.g., (label1, value1), and requires the data type of label1 to be TEXT. For the SELECT query, SQLite returns an error, which states that the json_object function takes labels that must be TEXT type. However, the label in the json_object function is indeed a TEXT column $c1$. We report this bug to SQLite developers, who explain that the constant propagation optimization causes this problem. SQLite suffers from "premature evaluation" of the json_object function in this bug. Specially, SQLite transforms predicate φ into (NULL == c1) AND json_object(NULL, NULL), and calculates json_object(NULL, NULL) in predicate φ without checking (NULL == c1).

```
CREATE TABLE t1 (c1 TEXT);
INSERT INTO t1 VALUES ('a'); -- r1
1. SELECT c1 FROM t1 WHERE 1 << c0;
    -- Fetch r1
    -- Warning|1292|evaluation failed:
        Truncated incorrect INTEGER value: 'a'
2. UPDATE t1 SET c1 = 'b' WHERE 1 << c0;
    -- Update no row
    -- Error|1292|Truncated incorrect INTEGER
        value: 'a'
3. DELETE FROM t1 WHERE 1 << c0;
    -- Delete no row
    -- Error|1292|Truncated incorrect INTEGER
        value: 'a'
```

Listing 6. TiDB#31391. The SELECT query raises an incorrect warning message.

Incorrect warning message. Listing 6 shows a bug TiDB#31391[9], in which the SELECT query raises a warning with an incorrect warning message. According to the error reference manual[10], the warning message format for the code 1292 is "Truncated incorrect %s value: '%s'". "evaluation failed: " should not appear in the SELECT query's warning message. This bug illustrates that cross-validating the evaluation of a predicate among SELECT, UPDATE and DELETE queries helps equip DQE with the ability to check the correctness of warnings.

```
CREATE TABLE t1 (c1 INT);
INSERT INTO t1 VALUES (1); -- r1
1. SELECT * FROM t1 WHERE POW(0, -1);
    -- Fetch empty result
    --
2. UPDATE t1 SET c1=2 WHERE POW(0, -1);
    -- Update no row
    -- Error|1690|DOUBLE value is out of range
        in 'pow(0,-(1))'
3. DELETE FROM t1 WHERE POW(0, -1);
    -- Delete no row
    -- Error|1690|DOUBLE value is out of range
        in 'pow(0,-(1))'
```

Listing 7. TiDB#33292. The SELECT query triggers incorrect functionality in the SHOW WARNINGS command.

Others. Listing 7 shows a bug TiDB#33292[11], in which the diagnostic command SHOW WARNINGS fails to return the error raised by the SELECT query. The SHOW WARNINGS command is a diagnostic command that returns warnings or errors resulting from the current query execution [28]. Because we use the SHOW WARNINGS command to obtain a query's raised warning in TiDB, DQE reports this bug due to a missing warning in the SELECT query. DQE find another two bugs that also happen in the SHOW WARNINGS command.

F. Not A Bug

In this section, we list a representative bug that is classified as not a bug in MySQL.

```
CREATE TABLE t1 (c1 FLOAT);
INSERT INTO t1 VALUES (1); -- r1
CREATE UNIQUE INDEX i1 ON t1 (c1 DESC);
1. SELECT * FROM t1 WHERE ('a'|1) BETWEEN 0 AND
    c1;
    -- Fetch r1
    -- Warning|1292|Truncated incorrect INTEGER
        value: 'a'
    -- Warning|1292|Truncated incorrect INTEGER
        value: 'a'
    -- Warning|1292|Truncated incorrect INTEGER
        value: 'a'
2. UPDATE t1 SET c1='b' WHERE ('a'|1) BETWEEN 0
    AND c1;
    -- Update no row
    -- Error|1292|Truncated incorrect INTEGER
        value: 'a'
    -- Warning|1292|Truncated incorrect INTEGER
        value: 'a'
3. DELETE FROM t1 WHERE ('a'|1) BETWEEN 0 AND c1
    ;
    -- Delete no row
    -- Error|1292|Truncated incorrect INTEGER
        value: 'a'
    -- Warning|1292|Truncated incorrect INTEGER
        value: 'a'
```

Listing 8. MySQL#106407. The SELECT query raises duplicate warnings.

[8]https://sqlite.org/forum/forumpost/12638a0aea0602a8

[9]https://github.com/pingcap/tidb/issues/31391

[10]https://dev.mysql.com/doc/mysql-errors/5.7/en/server-error-reference.html

[11]https://github.com/pingcap/tidb/issues/33292

Listing 8 shows a bug MySQL#106407[12], in which the same warning or error appears multiple times on an indexed column $r1$. Table $t1$ consists of a FLOAT row with value 0. There is an index built on column $c1$. The predicate φ is ('a' | 1) BETWEEN 0 AND c1. For the `SELECT` query, MySQL converts 'a' to an `INT` value 0, and then performs bitwise OR operation with 1, which result is 1, and finally checks whether the result is in the range of 0 and the value of column $c1$. Because the value of column $c1$ is 1, predicate φ is evaluated to `TRUE`. Therefore, the `SELECT` query returns row $r1$, but raises three same warnings. The `UPDATE` and `DELETE` queries raise a warning and an error (with the same warning code and message). We think there should be no duplicate warnings or errors because there is only one row. We report it to MySQL developers who confirm this issue, but explain that *"duplicate warnings, which are all identical, do not constitute a bug"*.

V. Discussion

Limitations. DQE has some limitations on detecting logic bugs. First, DQE faces the same problem as differential testing. DQE cannot detect a logic bug that occurs in all the three `SELECT`, `UPDATE` and `DELETE` queries. Second, DQE only supports common operations and functions supported by `SELECT`, `UPDATE` and `DELETE` queries. For each query's specific operations and functions, DQE cannot compare its execution results with other queries' execution results. For example, DQE cannot support aggregate-based functions, window functions and `GROUP BY` that are only used in `SELECT` queries. Third, DQE cannot support non-deterministic functions, e.g., `RAND` function that returns a random value in a query.

Extend to other DBMSs. The core idea of DQE is simple but rather applicable to other DBMSs, because most DBMSs support data manipulation specified by predicates. We expect that the key insight of DQE could be used in many database systems in DB-engine ranking [17]. We list some of them as follows. (1) Graph database systems, e.g., Neo4j [35], Microsoft Azure Cosmos DB [36], and TigerGraph [37]. (2) Key-value stores, e.g., Redis [38], Amazon DynamoDB [39], and Hazelcast [40]. (3) Document stores, e.g., MongoDB [41], CouchDB [42], and Google Cloud Datastore [43].

VI. Related Work

Differential testing of DBMSs. Differential testing [44] is effective to test DBMSs without facing the test oracle problem. The core idea behind differential testing is by feeding the same input to many functionally identical systems and comparing their outputs to detect bugs. There are many existing works applying differential testing on DBMSs [8], [30]–[32], [45]–[49]. RAGS [8] executes the same `SELECT` query on different DBMSs and observes discrepancies in their query results. DT2 [46] feeds a group of transactions into multiple DBMSs to detect transaction bugs. Grand [45] and RD2 [49] apply differential testing on graph database systems. APOLLO [30] feeds the same `SELECT` query into two different versions of the same DBMS to detect performance bugs. SparkFuzz [32] validates a query result with a reference DBMS (e.g., PostgreSQL) or with a different Spark version. We develop a novel differential testing approach, which executes `SELECT`, `UPDATE` and `DELETE` queries with the same predicate in a DBMS to detect logic bugs.

Database and SQL query generation. One key component of automatic testing is an automatic input generator. Database and SQL query generation have been widely explored by existing works [21]–[26], [29], [50]–[57]. SQLsmith [29] is an open source random SQL query generator, which is inspired by Csmith [58]. Go-randgen [54] can generate various SQL queries based on input SQL grammar. SQLRight [55] is a mutation-based SQL query generator, in which an intermediate representation is designed to perform mutations guided by coverage feedback. A better database and SQL query generation might improve the efficiency of our work in detecting logic bugs.

Test oracles of DBMSs. Test oracles are the key to reveal DBMS bugs. ADUSA [26] uses Alloy [59], an open source language and analyzer, to analyze the expected query result of a given `SELECT` query. PQS [9] synthesizes a `SELECT` query, which is computed to fetch a randomly-selected pivot row, and checks whether the pivot row is contained in its query result. NoREC [10] rewrites a `SELECT` query as an equivalent one that the DBMS cannot optimize, and compares their results. TLP [11] leverages the ternary property of predicate evaluation, where the evaluation result is one of `TRUE`, `FALSE` and `NULL`, to partition a `SELECT` query into three partitioning queries, whose combined query results are equal to the original query's query result. Troc [60] proposes how to build a test oracle for a pair of transactions. Our work proposes a new test oracle for DBMS testing, and is complementary to existing approaches.

VII. Conclusion

Logic bugs in `UPDATE` and `DELETE` queries can cause severer consequences, e.g., incorrect database states, and have not been tackled by existing approaches. In this paper, we propose a novel and general approach DQE to effectively detect logic bugs in `SELECT`, `UPDATE` and `DELETE` queries. We evaluate DQE on five widely-used DBMSs, i.e., MySQL, MariaDB, TiDB, CockroachDB and SQLite. In total, we have detected 41 previously-unknown logic bugs in these DBMSs. We expect that the generality of DQE can help improve the reliability of DBMSs.

Acknowledgments

This work was partially supported by National Key R&D Program of China (2021YFB1716000), National Natural Science Foundation of China (62072444), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences.

[12]https://bugs.mysql.com/bug.php?id=106407

REFERENCES

[1] "MySQL homepage," https://www.mysql.com, 2022.
[2] "MariaDB homepage," https://mariadb.org/, 2022.
[3] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang, "TiDB: A Raft-based HTAP database," *Proceedings of the VLDB Endowment (VLDB)*, vol. 13, no. 12, pp. 3072–3084, 2020.
[4] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis, "CockroachDB: The resilient Geo-distributed SQL database," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2020, pp. 1493–1509.
[5] "SQLite homepage," https://www.sqlite.org/index.html, 2022.
[6] D. D. Chamberlin and R. F. Boyce, "SEQUEL: A structured english query language," in *Proceedings of ACM SIGFIDET Workshop on Data Description, Access and Control*, 1974, pp. 249–264.
[7] "MySQL customers by industry," https://www.mysql.com, 2022.
[8] D. R. Slutz, "Massive stochastic testing of SQL," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 1998, pp. 618–622.
[9] M. Rigger and Z. Su, "Testing database engines via pivoted query synthesis," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 667–682.
[10] ——, "Detecting optimization bugs in database engines via non-optimizing reference engine construction," in *Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 1140–1152.
[11] ——, "Finding bugs in database systems via query partitioning," in *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, vol. 4, 2020.
[12] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.
[13] B. Ding, S. Das, W. Wu, S. Chaudhuri, and V. Narasayya, "Plan Stitch: Harnessing the best of many plans," *Proceedings of the VLDB Endowment (VLDB)*, vol. 11, no. 10, pp. 1123–1136, 2018.
[14] T. Neumann and B. Radke, "Adaptive optimization of very large join queries," in *Proceedings of International Conference on Management of Data (SIGMOD)*, 2018, pp. 677–692.
[15] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao, "Towards a learning optimizer for shared clouds," *Proceedings of the VLDB Endowment (VLDB)*, vol. 12, no. 3, pp. 210–222, 2018.
[16] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," *Proceedings of the VLDB Endowment (VLDB)*, vol. 12, no. 11, pp. 1705–1718, 2019.
[17] "DB-Engines ranking," https://db-engines.com/en/ranking, 2023.
[18] "Most widely deployed and used database engine," https://www.sqlite.org/mostdeployed.html, 2022.
[19] "Database topic in GitHub," https://github.com/topics/database, 2023.
[20] "Unexpected delete when data truncation," https://jira.mariadb.org/browse/MDEV-27885, 2022.
[21] A. Neufeld, G. Moerkotte, and P. C. Lockemann, "Generating consistent test data: Restricting the search space by a generator formula," *Proceedings of the VLDB Endowment (VLDB)*, vol. 2, no. 2, pp. 173–214, 1993.
[22] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1994, pp. 243–252.
[23] N. Bruno and S. Chaudhuri, "Flexible database generators," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2005, pp. 1097–1107.
[24] K. Houkjær, K. Torp, and R. Wind, "Simple and realistic data generation," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2006, pp. 1243–1246.
[25] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu, "QAGen: Generating query-aware test databases," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2007, pp. 341–352.
[26] S. Abdul Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid, "Query-aware test generation using a relational constraint solver," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008, pp. 238–247.
[27] "SQLancer homepage," https://github.com/sqlancer/sqlancer, 2022.
[28] "SHOW WARNINGS statement," https://dev.mysql.com/doc/refman/8.0/en/show-warnings.html., 2022.
[29] "SQLsmith," https://github.com/anse1/sqlsmith, 2015.
[30] J. Jung, H. Hu, J. Arulraj, T. Kim, and W. Kang, "APOLLO: Automatic detection and diagnosis of performance regressions in database systems," *Proceedings of the VLDB Endowment (VLDB)*, vol. 13, no. 1, pp. 57–70, 2019.
[31] X. Liu, Q. Zhou, J. Arulrai, and A. Orso, "Automatic detection of performance bugs in database systems using equivalent queries," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2022, pp. 225–236.
[32] B. Ghit, N. Poggi, J. Rosen, R. Xin, and P. Boncz, "SparkFuzz: Searching correctness regressions in modern query engines," in *Proceedings of the Workshop on Testing Database Systems (DBTest)*, 2020.
[33] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov, "Evaluating the "small scope hypothesis"," in *Proceedings of ACM Symposium on the Principles of Programming Languages (POPL)*, vol. 2, 2003.
[34] "Type conversion in expression evaluation," https://dev.mysql.com/doc/refman/5.7/en/type-conversion.html, 2022.
[35] "Neo4j homepage," https://neo4j.com/, 2022.
[36] "Azure cosmos DB," https://azure.microsoft.com/en-us/products/cosmos-db/, 2023.
[37] "TigerGraph," https://www.tigergraph.com/, 2023.
[38] "Redis homepage," https://redis.io/, 2022.
[39] "Amazon DynamoDB," https://aws.amazon.com/cn/dynamodb/, 2023.
[40] "Hazelcast," https://hazelcast.com/, 2023.
[41] "MongoDB," https://www.mongodb.com/, 2022.
[42] "Apache CouchDB," https://couchdb.apache.org/, 2023.
[43] "Datastore," https://cloud.google.com/datastore, 2023.
[44] W. M. McKeeman, "Differential testing for software," *DIGITAL TECHNICAL JOURNAL*, vol. 10, pp. 100–107, 1998.
[45] Y. Zheng, W. Dou, Y. Wang, Z. Qin, L. Tang, Y. Gao, D. Wang, W. Wang, and J. Wei, "Finding bugs in Gremlin-based graph database systems via randomized differential testing," in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022, pp. 302–313.
[46] Z. Cui, W. Dou, Q. Dai, J. Song, W. Wang, J. Wei, and D. Ye, "Differentially testing database transactions for fun and profit," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.
[47] J. Fu, J. Liang, Z. Wu, M. Wang, and Y. Jiang, "Griffin: Grammar-free DBMS fuzzing," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.
[48] W. Lin, Z. Hua, L. Zhang, and T. Xie, "GDiff: Automated differential performance testing for graph database systems," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2023.
[49] R. Yang, Y. Zheng, L. Tang, W. Dou, W. Wang, and J. Wei, "Randomized differential testing of RDF stores," in *Proceedings of International Conference on Software Engineering (ICSE Demo)*, 2023.
[50] J. Ba and M. Rigger, "Testing database engines via query plan guidance," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2023.
[51] Z. Jiang, J. Bai, and Z. Su, "DynSQL: Stateful fuzzing for database management systems with complex and valid SQL query generation," in *Proceedings of USENIX Security Symposium (USENIX Security)*, 2023.
[52] Z. Hua, W. Lin, L. Ren, Z. Li, L. Zhang, W. Jiao, and T. Xie, "GDsmith: Detecting bugs in Cypher graph database engines," 2023.
[53] M. Kamm, M. Rigger, C. Zhang, and Z. Su, "Testing graph database engines via query partitioning," in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023.
[54] "go-randgen," https://github.com/pingcap/go-randgen, 2020.
[55] Y. Liang, S. Liu, and H. Hu, "Detecting logical bugs of DBMS with Coverage-based guidance," in *Proceedings of USENIX Security Symposium (USENIX Security)*, 2022, pp. 4309–4326.
[56] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, "SQUIRREL: Testing database management systems with language validity and coverage feedback," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020, pp. 58–71.

[57] M. Wang, Z. Wu, X. Xu, J. Liang, C. Zhou, H. Zhang, and Y. Jiang, "Industry practice of Coverage-guided enterprise-level DBMS fuzzing," in *Proceedings of IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 328–337.

[58] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 283–294.

[59] "Alloy," https://alloytools.org/, 2022.

[60] W. Dou, Z. Cui, Q. Dai, J. Song, D. Wang, Y. Gao, W. Wang, J. Wei, L. Chen, H. Wang, H. Zhong, and T. Huang, "Detecting isolation bugs via transaction oracle construction," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2023.