# Detecting Isolation Bugs via Transaction Oracle Construction

Wensheng Dou*†‡§, Ziyu Cui*†, Qianwang Dai*†, Jiansen Song*†, Dong Wang*†, Yu Gao*†, Wei Wang*†‡§
Jun Wei*†‡§, Lei Chen¶, Hanmo Wang¶, Hua Zhong*†, Tao Huang*†
*State Key Lab 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
¶Inspur Software Group Co., Ltd.
*{wsdou, cuiziyu20, daiqianwang19, songjiansen20, wangdong18, gaoyu15, wangwei, wj, zhonghua, tao}@otcaix.iscas.ac.cn
¶{chen.leilc, wanghanmo}@inspur.com

***Abstract*—Transactions are used to maintain the data integrity of databases, and have become an indispensable feature in modern Database Management Systems (DBMSs). Despite extensive efforts in testing DBMSs and verifying transaction processing mechanisms, isolation bugs still exist in widely-used DBMSs when these DBMSs violate their claimed transaction isolation levels. Isolation bugs can cause severe consequences, e.g., incorrect query results and database states.**

**In this paper, we propose a novel transaction testing approach, *Transaction oracle construction* (*Troc*), to automatically detect isolation bugs in DBMSs. The core idea of Troc is to decouple a transaction into independent statements, and execute them on their own database views, which are constructed under the guidance of the claimed transaction isolation level. Any divergence between the actual transaction execution and the independent statement execution indicates an isolation bug. We implement and evaluate Troc on three widely-used DBMSs, i.e., MySQL, MariaDB, and TiDB. We have detected 5 previously-unknown isolation bugs in the latest versions of these DBMSs.**

***Index Terms*—Database system, transaction, isolation, oracle**

## I. Introduction

Database Management Systems (DBMSs), e.g., MySQL [1], MariaDB [2], PostgreSQL [3], TiDB [4], and CockroachDB [5], are widely used in many applications for efficiently storing and retrieving data with Structured Query Language (SQL) [6]. DBMSs treat the integrity of data as one of the most important promises, and utilize transactions to maintain the data integrity [7], [8]. In DBMSs, a transaction is a logical unit of work that consists of one or more database operations, i.e., SQL statements. DBMSs should ensure that all operations in a transaction are executed as a whole.

To ensure consistency of DBMSs, transactions should be executed in isolation from each other. Intermediate results from simultaneously executed transactions should not be visible to each other. However, stronger isolation among transactions can degrade DBMSs' performance more. Therefore, DBMSs usually provide multiple isolation levels, e.g., `Read Uncommitted`, `Read Committed`, `Repeatable Read`, and `Serializable` [9]–[11].

Wensheng Dou and Lei Chen are the corresponding authors.

To efficiently support different transaction isolation levels, DBMSs adopt many complex transaction processing mechanisms, e.g., lock-based concurrency control [12], [13], Multi-Version Concurrency Control (MVCC) [14]–[17], and Optimistic Concurrency Control (OCC) [18], [19]. Design flaws or buggy implementations of these mechanisms can violate their claimed isolation levels [20], i.e., causing *isolation bugs*. These bugs can lead to weaker isolation levels than claimed, and result in incorrect query results and database states.

To verify whether a DBMS actually provides the isolation guarantee as it claims, existing approaches [21]–[23] *carefully design specific transactions* against the DBMS, record a history of how these transactions are completed, and analyze the history to identify isolation bugs. These approaches, e.g., Cobra [22] and Elle [23], usually adopt a simple $key\text{-}value$ data model, and read and write *one row indexed by a key* with $read(key)$ and $write(key)$, respectively. Specially, the $value$ in Elle's data model [23] should be accumulative, e.g., AppendList. However, modern DBMSs, e.g., MySQL and TiDB, usually adopt a much more complex *relational* data model. They support many complex data structures (e.g., various data types, primary keys, and indexes), and access data with complex interface (e.g., reading / writing multiple rows through SQL). This introduces unique challenges to ensure the correctness of transaction processing mechanisms. Existing verification approaches cannot be generalized to verify transactions that utilize these features.

Automatic database testing approaches [24]–[28] can support these complex features in modern DBMSs, and have been proved as an effective technique to detect bugs in DBMSs. A key challenge for automatic database testing is to construct an effective test oracle, which can detect whether a DBMS behaves correctly. Existing approaches can construct some test oracles for *single queries* (i.e., SELECT statements). For example, SQLancer constructs the query partitioning oracle [27] and the containment oracle [26] for a *single query*. However, these testing approaches do not involve transactions, and cannot construct a test oracle for concurrent transactions at a certain isolation level, thus failing to detect isolation bugs.

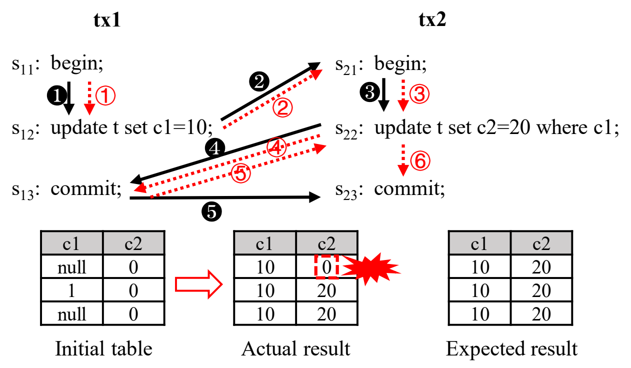

Fig. 1. An illustrative example that triggers a `serious` isolation bug in MySQL [29]. $tx1$ and $tx2$ are executed at `Read Committed` isolation level under the pessimistic transaction model. Black solid arrows show the submitted order, and red dotted arrows show the execution order.

We use a real-world isolation bug to explain why existing approaches cannot effectively detect isolation bugs in modern DBMSs. Fig. 1 shows a test case that triggers a `serious` isolation bug that we found in widely-used MySQL. This bug occurs at `Read Committed` isolation level under the pessimistic transaction model. In this test case, two transactions, i.e., $tx1$ and $tx2$, act on the initial table (the left table in Fig. 1). $tx1$ updates column $c1$ with value 10, and $tx2$ updates column $c2$ with value 20 where $c1$ is not `NULL`. The statements in $tx1$ and $tx2$ are submitted to MySQL in the order of black solid arrows, i.e., $s_{11} \rightarrow s_{12} \rightarrow s_{21} \rightarrow s_{22} \rightarrow s_{13} \rightarrow s_{23}$. Because $s_{22}$ in $tx2$ tries to update the same rows as $s_{12}$ in $tx1$, $s_{22}$ will be blocked by MySQL. After $tx1$ commits ($s_{13}$), $s_{22}$ in $tx2$ will be resumed. The red dotted arrows show the actual execution order, i.e., $s_{11} \rightarrow s_{12} \rightarrow s_{21} \rightarrow s_{22}(blocked) \rightarrow s_{13} \rightarrow s_{22}(resumed) \rightarrow s_{23}$. After $tx1$ and $tx2$ complete, the expected database state is shown in the right table, in which all data in column $c1$ are 10, and all data in column $c2$ are 20. However, the actual execution obtains an incorrect database state shown in the middle table, in which column $c2$ in the first row (with value 0) is not changed as expected. We reported this bug to MySQL developers, and they classified it as `serious`. However, MySQL developers have not figured out why this bug occurs yet.

Revealing the above isolation bug is challenging. We lack a transaction oracle to judge whether the result of the actual transaction execution is correct. Existing approaches are ineffective in detecting this bug. Automatic database testing approaches like SQLancer [26]–[28] detect logic bugs for single queries (i.e., SELECT statements) without considering concurrent transactions, and cannot be applied on other statements, e.g., UPDATE statements in $s_{12}$ and $s_{22}$. Transaction verification approaches, e.g., Cobra [22] and Elle [23], use read / write operation histories based on a key-value data model, and cannot handle this table structure (without a key) and multiple rows accessed in one SQL statement (e.g., $s_{12}$).

To effectively detect isolation bugs in modern DBMSs, we propose a novel and general transaction testing approach, *Transaction oracle construction* (*Troc*). The core idea is to solve the oracle problem of a pair of concurrent transactions by decoupling them into a series of independent statements, each of which can be executed on its own database view without its original transaction context. Specially, for each statement $stmt$ in a given submitted order of two transactions $tx1$ and $tx2$, we first automatically construct $stmt$'s own database view under the guidance of the given isolation level. Based on $stmt$'s own database view, we analyze whether $tx1$ and $tx2$ conflict on $stmt$, and further infer its expected execution result. Finally, we compare the actual transaction execution results with these constructed independent statement execution results, and report an isolation bug if they diverge. By decoupling concurrent transactions into a series of independent statements and executing these statements independently, we can avoid performing DBMSs' complex transaction processing mechanisms in our oracle construction. Therefore, Troc can reveal bugs introduced in them.

To demonstrate the effectiveness of Troc, we implement it for three widely-used production-level DBMSs, i.e., MySQL [1], MariaDB [2] and TiDB [4], and test all their isolation levels under the pessimistic transaction model. In total, we have detected 12 bugs, in which, 10 bugs are isolation bugs and the remaining 2 bugs are transaction-related. We have reported these bugs to developers, and 7 bugs (5 isolation bugs and 2 transaction bugs) have been confirmed as new bugs, and the remaining 5 isolation bugs are classified as duplicate. One newly detected isolation bug has been fixed by developers. For these 7 newly detected bugs, 4 bugs leave the database into an incorrect state, and 3 bugs cause incorrect query results. We have made Troc publicly available at https://github.com/tcse-iscas/Troc.

In summary, we make the following contributions.

- We propose an oracle construction approach for testing a pair of concurrent transactions in DBMSs.
- We propose Troc, a novel and general transaction testing technique for detecting isolation bugs in DBMSs.
- We implement Troc and apply it on three widely-used DBMSs, i.e., MySQL, MariaDB and TiDB. Our evaluation has revealed 5 previously-unknown isolation bugs.

## II. Preliminaries

**SQL and transaction.** We primarily target DBMSs based on the relational data model [30], i.e., relational DBMSs. Most widely-used DBMSs like Oracle, MySQL, PostgreSQL and TiDB are this type of DBMSs. Users usually interact with relational DBMSs through Structured Query Language (SQL) [6]. In relational DBMSs, a transaction is a logical unit of work that is executed as a whole. A transaction starts with a `BEGIN` statement, contains a list of SQL statements, and ends with a `COMMIT` or `ROLLBACK` statement.

**Target DBMSs.** We focus on three widely-used production-level DBMSs, i.e., MySQL, MariaDB, and TiDB. According to DB-Engines Ranking [31] and GitHub stars [32] in Table I, they are ones of the most widely-used and popular DBMSs. All these DBMSs have been maintained for a long time. Note

TABLE I
TARGET DBMSS IN OUR STUDY

| DBMS | DB-Engines | GitHub stars | Release | Isolation levels |
|---|---|---|---|---|
| MySQL | 2 | 8.7K | 1995 | RU, RC, RR, SER |
| MariaDB | 13 | 4.7K | 2009 | RU, RC, RR, SER |
| TiDB | 108 | 33.3K | 2017 | RC, RR |

that, MySQL and MariaDB are traditional relational DBMSs, while TiDB is an open-source NewSQL distributed DBMS.

MySQL and MariaDB adopt the pessimistic transaction model, in which, if a SQL statement in transaction $tx1$ conflicts with another transaction $tx2$, $tx1$ will be blocked and continue its execution after $tx2$ has completed, i.e., rolled back or committed. TiDB supports both pessimistic and optimistic transaction models and adopts the pessimistic transaction model by default. In TiDB's optimistic transaction model, transaction $tx1$ can be aborted (decided by some strategies) if $tx1$ conflicts with another transaction $tx2$. In our work, we mainly explain Troc under the pessimistic transaction model, which is implemented by all the three target DBMSs.

**Transaction isolation levels.** DBMSs usually provide multiple isolation levels to make a tradeoff between consistency and performance. The stronger an isolation level is, the more likely concurrent transactions are sequentially executed. There are more than 10 isolation levels in existing works [9]–[11], [33]–[37]. Here, we mainly explain the isolation levels under the pessimistic transaction model in our target DBMSs.

As shown in Table I, these DBMSs support different isolation levels under the pessimistic transaction model. MySQL and MariaDB support four isolation levels [38], [39], i.e., `Read Uncommitted`, `Read Committed`, `Repeatable Read`, and `Serializable`. TiDB supports two isolation levels, i.e., `Read Committed` and `Repeatable Read` [40]. In our work, we mainly explain Troc at the above four isolation levels. We simply explain them as follows.

- `Read Uncommitted (RU)`. RU in MySQL and MariaDB prevents the data modified by an uncommitted transaction from being overwritten by another transaction (i.e., dirty writes), and allows transaction $tx1$ to see transaction $tx2$'s modifications no matter $tx2$ is committed or not.
- `Read Committed (RC)`. RC in MySQL, MariaDB and TiDB requires that transaction $tx1$ can see transaction $tx2$'s modifications if $tx2$ has been committed, and all modifications of $tx2$ are visible to $tx1$.
- `Repeatable Read (RR)`. RR in MySQL, MariaDB and TiDB requires the rows read by transaction $tx1$ to be from the same snapshot for transaction $tx1$, i.e., snapshot read. RR further requires that $tx1$ can read its own writes, and $tx1$'s writes can see the committed rows by another transaction.
- `Serializable (SER)`. SER in MySQL and MariaDB is the strongest isolation level that DBMSs aim to achieve, and requires that the execution of $tx1$ and $tx2$ is equivalent to the execution in their certain serial order.

**Algorithm 1:** Transaction test protocol under the pessimistic transaction model

```
Input: $tx1$, $tx2$, $subOrder$
1  for $i \leftarrow 1; i \leq subOrder.length; i++$ do
2      $stmt \leftarrow subOrder[i]$
3      $curTx \leftarrow stmt.transaction$
4      $otherTx \leftarrow curTx = tx1\ ?\ tx2\ :\ tx1$
5      if $curTx.blocked$ then
6          $curTx.blockedStmts.add(stmt)$
7          continue
8      $\underline{execState \leftarrow curTx.submit(stmt)}$
9      if $execState.reportDeadlock()$ then
10         $curTx.blocked \leftarrow True$
11         break
12     if $execState.blocked$ then
13         $curTx.blocked \leftarrow True$
14     if $stmt.type = COMMIT \mid ROLLBACK$ then
15         $otherTx.blocked \leftarrow False$
16         while $otherTx.blockedStmts \neq \emptyset$ do
17             $s \leftarrow$
                 $otherTx.blockedStmts.removeFirst()$
18             $\underline{execState \leftarrow otherTx.submit(s)}$
19 if $!curTx.blocked\ \wedge\ !otherTx.blocked$ then
20     return $\underline{database.state}$
```

## III. APPROACH

We propose <u>Tr</u>ansaction <u>o</u>racle <u>c</u>onstruction (*Troc*), to automatically detect isolation bugs in DBMSs under the pessimistic transaction model. In the following, we first explain the transaction test protocol in Troc (Section III-A), and then explain how Troc works (Section III-B-III-F).

### A. Transaction Test Protocol

For a group of transactions $\{tx1, tx2, tx3, ..., txn\}$ under the pessimistic transaction model, we can simultaneously submit them to a target DBMS. However, it is challenging to infer their expected execution results due to the following reasons. First, for two simultaneously submitted SQL statements from two transactions, which statement is scheduled first by the DBMS is nondeterministic. Second, if more than one transaction, e.g., $tx1$ and $tx2$, are blocked by the same transaction $tx$, after $tx$ completes, which transaction ($tx1$ or $tx2$) is resumed and scheduled next is nondeterministic.

To address the above challenges, we focus on a relatively simple transaction test scenario. First, a transaction test case involves a pair of transactions, i.e., $tx1$ and $tx2$. Second, we submit the SQL statements in $tx1$ and $tx2$ to the target DBMS one by one, by following a certain submitted order. Under the pessimistic transaction model, if the submitted order of each statement in $tx1$ and $tx2$ are determined, we can only obtain *one* deterministic execution order on the target DBMS, since we can certainly know which statement is scheduled next.

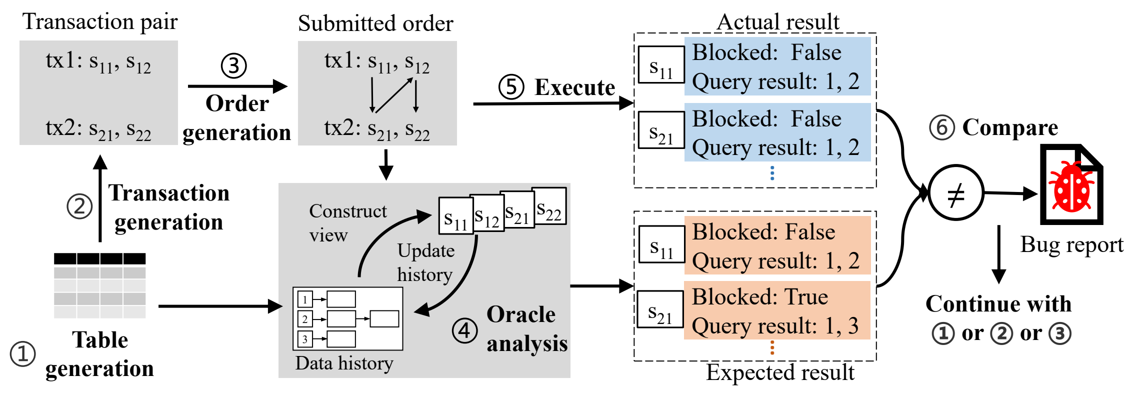

Fig. 2. Troc's workflow.

Algorithm 1 illustrates our transaction test protocol under the pessimistic transaction model. In this protocol, we submit the statements in the submitted order $subOrder$ to the target DBMS one by one. If a deadlock is reported, we set $curTx$ as blocked, and stop submitting the following statements (Line 9-11). If transaction $curTx$ is blocked at a statement (Line 12-13), since $curTx$ cannot be unblocked before the other transaction $otherTx$ terminates, we will block $curTx$'s remaining statements (Line 5-7) until it is resumed by transaction $otherTx$ (Line 14-15). Once $curTx$ is resumed, all its blocked statements will be executed first (Line 16-18). After two transactions complete, we can retrieve the final database state if no deadlock is detected (Line 19-20).

## B. Troc Overview

Troc can automatically construct the transaction oracle for a submitted order of a pair of transactions $tx1$ and $tx2$ under the pessimistic transaction model, and then automatically detect isolation bugs in DBMSs. Our key insight is that, a SQL statement $stmt$ in a transaction $tx$ can be executed independently (i.e., $stmt$ is not executed in $tx$) on a specific database view that can be constructed based on $tx$'s isolation level, and returns the same result as its execution in $tx$. For example, in Fig. 1, the execution results of $tx1$ and $tx2$ at `Read Committed` should be equivalent to the results of executing $s_{12}$ and $s_{22}$ sequentially on the initial table. If the actual transaction execution diverges from the constructed independent statement execution, we detect an isolation bug.

Fig. 2 shows the detailed steps of Troc. First, we create a random table schema and populate the table with random data ①. Based on the generated table, we generate two transactions, each of which consists of a group of SQL statements ②, and then randomly generate a submitted order of the two transactions as a transaction test case ③. For the generated test case, we perform transaction oracle analysis to construct each SQL statement's expected execution results ④. We then submit the two transactions to the target DBMS in the same submitted order and record their actual execution results ⑤. If the actual execution results are different from the expected execution results, we detect an isolation bug in the target DBMS ⑥. For the next iteration, we can randomly continue with ① generating a new table, or ② generating a new pair of transactions, or ③ generating a new submitted order.

To construct the transaction oracle for a given submitted order of $tx1$ and $tx2$, we need to figure out how each statement is executed in this order, i.e., whether it is blocked, and what its execution result is. Specifically, we need to address three technical challenges. First, without understanding a SQL statement $stmt$'s semantics (e.g., obtaining $stmt$'s semantics through parsing $stmt$), how can we construct $stmt$'s own database view? Note that, $tx1$ and $tx2$ can have different database views at the same time. For example, $tx1$ and $tx2$ can only see their own snapshots at `Repeatable Read` isolation level. Second, without modeling DBMSs' inner lock behaviors, how can we judge whether a statement $stmt$ should be blocked? Note that, DBMSs usually adopt complex and non-standard lock implementations. Third, how can we obtain $stmt$'s expected execution result without $stmt$'s original transaction context?

We propose several novel techniques to address the above challenges. To address the first challenge, we assign a unique $rowId$ for each row in the database table (Section III-C). We further maintain a database change history for each row, which records all modifications made by SQL statements in $tx1$ and $tx2$ (Section III-E1). For each statement $stmt$ in transaction $tx$, we construct $stmt$'s database view (i.e., what data it can see) based on the database change history and $tx$'s isolation level (Section III-E2). For a SQL statement that updates / inserts / deletes data, we identify the modified data on $stmt$'s database view, and then update the database change history (Section III-E4). To address the second challenge, we perform a fine-grained row-based lock analysis based on $stmt$'s database view (Section III-E3). Our lock analysis is (almost) DBMS-agnostic, and can only lock necessary rows and indexes, which are commonly supported by DBMSs. If $stmt$ accesses common data accessed by the other transaction, $stmt$'s execution is blocked. To address the third challenge, we execute $stmt$ on its own database view, and obtain its expected result without $stmt$'s original transaction context.

Based on the above techniques, Troc can construct oracle for transactions with complex data structures and SQL in DBMSs, without manually building a reference model for SQL statements (e.g., SELECT and UPDATE). Troc treats database and SQL statement execution as a black box (e.g., without understanding SQL statements' concrete semantics by interpreting SQL statements), and does not limit data structures and SQL statements (except JOIN, UNION, sub queries and cross-table queries that we cannot support) for target DBMSs.

### *C. Database and SQL Statement Generation*

Random database [41]–[46] and SQL statement [24], [25], [47], [48] generation has been widely explored, and is not a contribution of this work. Troc can utilize existing database and SQL statement generation approaches. Troc's database and SQL statement generation mainly bases on SQLancer [49]. Here, we explain our database and SQL statement generation approach only for completeness.

For database generation, we use the `CREATE TABLE` statement to create a table with at most $maxCol$ (5 by default) columns. For each column, we randomly assign a column type and add column constraints, e.g., `PRIMARY KEY`, `NOT NULL` and `UNIQUE`. We further use the `CREATE INDEX` statement with randomly selected columns to add indexes on the table.

To populate random data into the generated database, we use at most $maxInsert$ `INSERT` statements to insert at most $maxInsert$ rows. We observe that isolation bugs can usually be triggered with a limited number of rows. By default, we set $maxInsert = 10$. Note that, the randomness can violate column constraints, and cause execution failures. For example, `UNIQUE` constraint prohibits the same value from appearing more than once in a column. If we cannot insert any data rows successfully, we discard the database.

**Unique rowId.** After database generation, we further add a column $rowId$ on the generated table. The column type of $rowId$ is set as `INTEGER`, and it has a `UNIQUE` constraint. After we insert data rows using `INSERT` statements, we assign a unique ID in a row if we find its $rowId$ is empty. Note that, the $rowId$ column is only used in our transaction oracle construction. We ensure that $rowId$ is not used in the following SQL statement and transaction generation.

```
1) SELECT columns FROM table WHERE condition
2) SELECT columns FROM table WHERE condition FOR
   SHARE
3) SELECT columns FROM table WHERE condition FOR
   UPDATE
4) UPDATE table SET (column = value)s WHERE condition
5) DELETE FROM table WHERE condition
6) INSERT INTO table(columns) VALUES values
```

Listing 1. SQL statements supported in Troc

For SQL statement generation, Troc generates six types of SQL statements in Listing 1. Since different DBMSs support different dialects, Troc randomly generates SQL statements based on the grammar of respective DBMSs, and constructs valid SQL statements specific to respective DBMSs. We use the database schema to generate valid column references if needed. For constants used in these SQL statements, e.g., inserted values, we randomly use one of the following two strategies. (1) We randomly generate a new value. (2) We randomly pick up a value from the corresponding column's value cache, which stores the generated values for the column.

Note that, we require that all statements in two transactions act on the same table. For now, Troc can support many complex data structures and SQL statements, e.g., primary keys, indexes, various data types, and conditions that are supported by SQLancer [49]. But, Troc cannot support JOIN, UNION, sub queries, etc. Multiple tables and `JOIN` can introduce complex cases for transaction oracle construction, we leave them as our future work.

### *D. Transaction Test Case Generation*

**Transaction generation.** We first randomly generate at most $txSize$ SQL statements in Listing 1 using the method illustrated in Section III-C, and add them into a transaction. We further append a start statement `BEGIN`, and an ending statement `COMMIT` or `ROLLBACK`, which are randomly chosen. $txSize$ is a configurable parameter that can be used to make a tradeoff between the transaction complexity and testing efficiency. In our work, we set $txSize = 10$ by default.

**Submitted order generation.** Given a transaction pair $tx1$ and $tx2$, we submit the SQL statements in $tx1$ and $tx2$ to DBMSs one by one according to the transaction test protocol in Algorithm 1. There are $N = C_{tx1.len+tx2.len}^{tx1.len}$ possible submitted orders. For example, if both $tx1$ and $tx2$ contain 10 statements, there are $C_{20}^{10} = 184,756$ possible submitted orders. To avoid being stuck in testing one transaction pair, we randomly generate one submitted order each time. The generation algorithm is simple. Each time, we randomly choose a transaction from $tx1$ and $tx2$, and append one statement from the chosen transaction into the submitted order.

### *E. Transaction Oracle Construction*

To analyze the execution of a statement $stmt$ in a transaction $tx$, we need to obtain the database view that $tx$ can see at statement $stmt$. We simply call this database view as $stmt$'s view. $stmt$'s view is decided by $stmt$'s type and $tx$'s isolation level that specifies whether other transactions' modifications are visible to $tx$.

Concurrent transactions executing on the same table can see different database views at the same time. To resolve this challenge, we first record the database change history, which stores the modification history of all rows in the database under test (Section III-E1 and Section III-E4). We further construct a statement's database view (Section III-E2) based on the database change history. Finally, we perform conflict analysis and obtain the expected execution results based on the constructed database view (Section III-E3 and Section III-E4).

*1) Database Change History:* The database change history records the modification process performed by a transaction test case. For each row, we store all its versions as a linked list, and each node in the list represents a version that shows how the row has been changed. Each version is represented by a triple $< data, tx, deleted >$, in which, $data$ contains the row data in current version, $tx$ records the transaction that

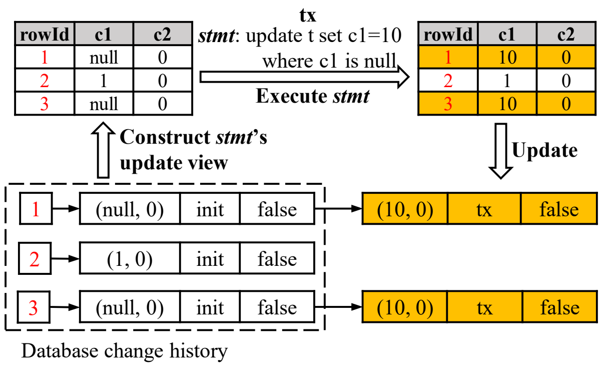

Fig. 3. Construct $tx$'s view and update database change history for statement $stmt$ on the initial table in Fig. 1.

creates the version, and *deleted* indicates whether the row has been deleted.

Fig. 3 illustrates an example how we initialize and maintain the database change history for the initial table in Fig. 1. For each row in the initial table, we construct a list indexed by its $rowId$, and add an initial version, e.g., $< (null, 0), init, false >$ for the first row with $rowId = 1$. The lists in the dotted box show the initial database change history. Note that, we use $tx = init$ for all the initial versions that are inserted by the database generation in Section III-C.

Assume that transaction $tx$ executes an update statement $stmt$ on the initial table, and no other transactions act on this table. We first construct $stmt$'s update view from the database change history. Note that what versions are visible for $stmt$ depends on $tx$'s isolation level and $stmt$'s type, and we will introduce the detailed construction strategy in Section III-E2. Since no other transactions in this case, $tx$ reads all latest versions in the database change history, and constructs $stmt$'s update view as the table in the top left corner in Fig. 3. Based on this view, we can perform conflict analysis to figure out whether $stmt$ is blocked (Section III-E3). In this example, $stmt$ can be directly executed, and obtains a new view (the table in the top right corner). We construct a new SQL statement `SELECT rowId FROM` $stmt$`.view WHERE` $stmt$`.condition` to obtain the updated rows, i.e., row 1 and 3 in Fig. 3. Finally, we update the database change history by attaching the versions of the two updated rows. Different kinds of statements have different update strategies, and we introduce them in Section III-E4.

*2) View Construction:* For a statement $stmt$ in transaction $tx1$ and $tx2$, $stmt$'s database view determines what data $stmt$ can see when $stmt$ is executed. $stmt$'s database view is decided by two factors: $stmt$'s statement type and the isolation level of $stmt$'s transaction. We do not need to interpret other parts of $stmt$, e.g., $stmt$'s `WHERE` condition, to construct $stmt$'s view. For easy presentation, we assume that $stmt$ belongs to $tx1$ in the following discussion, and all data in the initial table is written by transaction $init$.

We first summarize the following four data visibility rules, which are widely used in different DBMSs.

TABLE II
DATA VISIBILITY STRATEGIES IN MYSQL AND MARIADB

| Statement Type | Isolation Level | | | |
|---|---|---|---|---|
| | RU | RC | RR | SER |
| SELECT | V1, V2 | V1, V3 | V1, V4 | V1 |
| SELECT FOR SHARE | V1, V3 | V1, V3 | V1, V3 | V1 |
| SELECT FOR UPDATE | V1, V3 | V1, V3 | V1, V3 | V1 |
| UPDATE | V1, V3 | V1, V3 | V1, V3 | V1 |
| INSERT | V1, V3 | V1, V3 | V1, V3 | V1 |
| DELETE | V1, V3 | V1, V3 | V1, V3 | V1 |

- **V1.** The versions written by $tx1$ are visible to $stmt$. The versions written by transactions committed before $tx1$ starts are visible to $stmt$. For example, the versions written by $init$ are visible to $stmt$. If $tx2$ has been committed before $tx1$ starts, the versions written by $tx2$ are visible to $stmt$.
- **V2.** The versions written by $tx2$ are visible to $stmt$, no matter $tx2$ is committed or not.
- **V3.** The versions written by $tx2$ are visible to $stmt$ only if $tx2$ has been committed.
- **V4.** The committed versions before $tx1$ takes snapshot are visible to $stmt$. For example, the versions written by $tx2$ are visible to $stmt$ only if $tx2$ has been committed before $tx1$ takes snapshot.

DBMSs utilize different data visibility strategies for different isolation levels and SQL statements. Table II shows the detailed data visibility strategies for the four isolation levels used in MySQL and MariaDB. For example, if $stmt$ is a `SELECT` statement, all committed versions before $tx1$ performs snapshot (V4) and $tx1$'s writes (V1) are visible to $stmt$. These data visibility strategies can be found in DBMSs' documentations about transaction models [39], [50]. Note that, TiDB does not support `Read Uncommitted` and `Serializable` isolation levels [40]. But, the data visibility strategies for the two isolation levels in TiDB are the same as those used in MySQL.

Based on a DBMS's data visibility strategy, we can obtain the versions of each row in the database history that are visible to $stmt$ when $stmt$ is executed. Troc then traverses each row in the database history, and searches for its *latest* version $lv$ that is visible to $stmt$. If $lv.deleted$ is false, then we add $< lv.rowId, lv.data >$ into $stmt$'s database view. We further convert $stmt$'s database view into a database table. Thus, we can execute SQL statements through DBMSs on the table to perform the following conflict analysis (Section III-E3) and statement execution analysis (Section III-E4).

*3) Statement Conflict Analysis:* DBMSs usually adopt complex and non-standard lock behaviors, e.g., gap lock in MySQL. It is impractical to model the detailed lock behaviors in different DBMSs. Therefore, we perform a fine-grained row-based lock analysis in a conservative way, in which we only lock necessary rows and indexes based on current database states. Our lock analysis is DBMS-agnostic, and commonly supported by DBMSs. In the following discussion, we assume that $stmt$ belongs to $tx1$.

TABLE III
LOCK STRATEGIES IN MYSQL, MARIADB, AND TIDB

| Statement Type | MySQL & MariaDB | | | | TiDB | |
|---|---|---|---|---|---|---|
| | RU | RC | RR | SER | RC | RR |
| SELECT | - | - | - | S: R&I, P | - | - |
| SELECT FOR SHARE | S: R&I | S: R&I | S: R&I, P | S: R&I, P | S: R&I | S: R&I |
| SELECT FOR UPDATE | X: R&I | X: R&I | X: R&I, P | X: R&I, P | X: R&I | X: R&I |
| UPDATE | X: R&I | X: R&I | X: R&I, P | X: R&I, P | X: R&I | X: R&I |
| INSERT | X: R&I | X: R&I | X: R&I | X: R&I | X: R&I | X: R&I |
| DELETE | X: R&I | X: R&I | X: R&I, P | X: R&I, P | X: R&I | X: R&I |

S: Share lock. X: Exclusive lock. R&I: Row and index lock. P: Predicate lock.

**Fine-grained row-based lock analysis.** $stmt$ can lock a set of rows that it accesses, and a set of indexes (i.e., declared by `PRIMARY KEY` and `UNIQUE INDEX` in the database schema). For example, a statement inserting a row with primary key = 1 conflicts with another statement deleting a row with primary key = 1, even though they access different rows. We model $stmt$'s row and index lock as a triple $< type, lockRows, lockIndexes >$. $type$ denotes a lock's type, i.e., Share and Exclusive. $lockRows$ denotes a set of rows that are accessed by $stmt$. $lockIndexes$ denotes a set of $< index, value >$ that are accessed by $stmt$, in which $index$ is type of `UNIQUE INDEX` or `PRIMARY KEY` in the database schema.

We obtain $lockRows$ based on $stmt$'s statement type. If $stmt$ is type of `SELECT`, `UPDATE`, `DELETE` and `SELECT FOR SHARE / UPDATE`, it contains a condition expression. We construct a new query `SELECT rowId FROM` $stmt$`.view WHERE` $stmt$`.condition`, and obtain all $rowId$s that $stmt$ accesses. If $stmt$ is type of `INSERT`, after we execute $stmt$, we obtain the new inserted row's $rowId$, and put it in into $lockRows$.

For each row in $stmt$'s $lockRows$, we extract its `PRIMARY KEY` and `UNIQUE INDEX` columns, and their corresponding values to form $< index, value >$ pairs, and put them into $lockIndexes$. For `INSERT` and `UPDATE` statements, we also take the index values updated or inserted by the statement into consideration. We extract corresponding values from their statements and database states, and form $< index, value >$ pairs that they write, and put them into $lockIndexes$. Note that, for a composite unique index that consists of more than one column, if the value of one column is NULL, we will not add the corresponding $< index, value >$ into $lockIndexes$.

The above analysis cannot handle *predicate lock* specified by statements having conditions, e.g., `SELECT * FROM table WHERE c1>0 FOR SHARE` can lock all rows that satisfy `c1>0`. Assume that $stmt1$ in $tx1$ has a condition, and $stmt2$ belongs to $tx2$. First, if $stmt1$ and $stmt2$'s $lockRows$ or $lockIndexes$ overlap, $stmt1$'s predicate lock conflicts with $stmt2$. Second, if $stmt2$ (i.e., INSERT, UPDATE and DELETE) can modify the table, we check if $stmt2$ affects the execution of $stmt1$'s condition. We obtain $lockRows$ and $lockIndexes$ of $stmt1$ before and after executing $stmt2$ on $stmt1$'s database view, respectively, and check if they are the same. If not, $stmt2$'s execution can affect $stmt1$'s condition, and we consider $stmt1$'s predicate lock conflicts with $stmt2$. Predicate locks also have two types, i.e., Share and Exclusive.

**Conflict detection.** $stmt$'s required locks are decided by $stmt$'s statement type [51] and its isolation level. Table III summarizes lock strategies in MySQL, MariaDB and TiDB. We can see that MySQL and MariaDB adopt the same lock strategy, while TiDB adopts a slightly different lock strategy.

To judge whether $stmt1$ in $tx1$ and $stmt2$ in $tx2$ conflict, we first utilize Table III to extract their lock strategies based on their statement types and isolation levels. We consider $stmt1$ in $tx1$ and $stmt2$ in $tx2$ as conflict when they satisfy two conditions. (1) $stmt1$'s locks or $stmt2$'s locks are type of Exclusive. (2) Their $lockRows$ or $lockIndexes$ overlap, or their predicate locks conflict.

Note that, Troc's conflict analysis bases on row-based locks, and provides fine-grained lock analysis. DBMSs may lock more rows than our inferred results due to few reasons, e.g., range queries [52] and query optimizations [53]. Assume that a simple table has one column $pk$ as primary key, and the table contains two rows with values 1 and 10. At `Serializable` isolation level, we infer that the first row with value 1 is locked for a range query `SELECT * FROM table WHERE pk < 5`. However, due to gap lock in MySQL, this query can also lock the range [1, 10]. This makes our lock analysis incomplete, i.e., some statements are blocked in DBMSs but Troc infers that they should be executed. On the other hand, our lock analysis is sound, i.e., if Troc infers that a statement should be blocked, the statement must be blocked in DBMSs.

*4) Statement Execution Analysis:* When Troc decides that a statement $stmt$ in transaction $tx1$ should be blocked, we switch to transaction $tx2$ and continue performing the above analysis on its statements. If $stmt$ is decided not to be blocked, we execute it on its database view, and further update the database change history based on the new view. We explain the detailed strategies according to $stmt$'s statement type.

**SELECT, SELECT FOR SHARE / UPDATE.** We execute $stmt$ on its database view, and obtain its query result, which is used as $stmt$'s expected execution result.

**INSERT.** We directly execute $stmt$ on its view, and compare $stmt$'s view before and after $stmt$'s execution to obtain the newly inserted rows. We further construct a SQL statement `SELECT * FROM` $stmt$`.view WHERE rowId =` $iRow$ for each inserted row $iRow$ to obtain its inserted data. We then add a new list in the database change history for each newly inserted row. Each new list starts by the inserted row's $rowId$, and a row version that contains the inserted data.

**UPDATE.** We construct a new SQL statement `SELECT rowId FROM` *stmt*`.view WHERE` *stmt*`.condition` to obtain the updated rows. After we execute *stmt* on its view, we construct a new SQL statement `SELECT * FROM` *stmt*`.view WHERE rowId =` *uRow* for each updated row *uRow* to obtain its new data. We then add new versions at the tail of the database change history for each updated row.

**DELETE.** We construct a new SQL statement `SELECT rowId FROM` *stmt*`.view WHERE` *stmt*`.condition` to obtain the deleted rows. We then add a new version for each deleted row, in which its *deleted* field is set as *true*.

**ROLLBACK.** When `ROLLBACK` is executed, all data modifications made by transaction $tx1$ are aborted. We remove all versions in database change history whose $tx$ field is $tx1$.

### *F. Detecting Isolation Bugs*

Given a submitted order *subOrder* of transactions $tx1$ and $tx2$, we follow the test protocol in Algorithm 1, and submit each statement in *subOrder* to the target DBMS one by one. For each SQL statement, we obtain two aspects of results, i.e., (1) whether a statement is blocked (Line 8-13) and (2) the returned results for query statements (Line 8 and 18). After $tx1$ and $tx2$ complete, we also obtain (3) the final database state if no deadlock is detected (Line 19-20).

We analyze the statements in *subOrder* one by one, and compare each statement's actual execution result on the target DBMS and the expected execution result in the constructed transaction oracle. Note that, if either the actual result or the expected result reports a deadlock at statement *stmt*, we no longer compare its following statements in *subOrder*. For a statement *stmt*, we can detect three kinds of isolation bugs.

**Inconsistent blocking.** If *stmt* is blocked in the expected execution, but it is successfully executed in the actual execution, we report an isolation bug with inconsistent blocking. As discussed in Section III-E3, we adopt a fine-grained row-based lock analysis. It only locks necessary rows and indexes, while DBMSs may lock more due to range queries, etc. Therefore, if we infer that *stmt* is not blocked in the expected execution, but *stmt* is blocked in the actual execution, we can hardly say there is an isolation bug. To avoid reporting false positives, we do not report isolation bugs in such case.

**Incorrect query results.** If *stmt* is both executed (i.e., not blocked) in the actual execution and in the expected execution, we further compare their execution results, e.g., returned data. Any difference indicates an isolation bug.

**Incorrect final database states.** We compare the final database states only when both the actual execution and the expected execution follow the same order, and no deadlock is detected. Any difference indicates an isolation bug.

## IV. Evaluation

To demonstrate Troc's effectiveness, we evaluate Troc on widely-used DBMSs, and detect real-world isolation bugs. Our evaluation aims to address the following research questions:

- **RQ1:** How effective is Troc in finding real-world isolation bugs? (Section IV-B)
- **RQ2:** What bugs can Troc find in real-world DBMSs? (Section IV-C)

### *A. Experimental Methodology*

**Target DBMSs.** To demonstrate the effectiveness of Troc, we evaluate it on three widely-used DBMSs, i.e., MySQL, MariaDB, and TiDB. As shown in Table I, all these relational DBMSs provide good support for transactions, and are among the most widely-used DBMSs. MySQL is a standalone DBMS, and ranks on the top of the public ranking lists. MariaDB is an open-source fork of MySQL. TiDB is a representative of distributed NewSQL DBMS. For each target DBMS, we perform our testing on the latest release versions when we started this research, i.e., MySQL 8.0.25, MariaDB 10.5.12, and TiDB 5.2.0.

**Testing methodology.** The experiment is performed in an AliCloud server with 8 CPU core and 32 GiB memory. For MySQL and MariaDB, we allocate a Docker container and create a DBMS instance inside it. For TiDB, we test it in a distributed manner with 2 TiDB instances, 3 TiKV instances and 2 PD server instances. The whole experiment is performed automatically. Our experiment uses 10% CPU and 13.6 GiB memory at most.

For each testing iteration, we continuously run Troc on the target DBMSs until we find bugs. Once a bug is detected during testing, Troc produces a bug report, which contains its bug type, the constructed oracle as well as the actual execution result. Besides, it also contains all essential elements for reproducing the bug, i.e., the initial table, two transactions, the submitted order and isolation levels. If Troc cannot report a bug in two weeks, we stop the current testing iteration. We then inspect the reported bugs. For a reported bug, we first try to manually reproduce it and simplify the test case by removing statements and columns in the table that are not related to the bug. Since the bug is triggered at a specific isolation level, we further enumerate all possible isolation levels for checking whether the bug can manifest at other isolation levels, which can help understand the bug. At last, we summarize all information and report the bug to developers for feedbacks. In total, our experiment takes about two months.

### *B. Overall Bug Detection Results*

Table IV shows all the bugs detected by Troc in different DBMSs at different isolation levels (Column 2-5). Note that, a bug can be triggered at different isolation levels. We count the bugs at different isolation levels but having the same test inputs and similar symptoms as the same bug. Finally, we find 17 bugs in total, and 12 bugs are unique.

For the 12 unique bugs, 9 bugs are found at `Repeatable Read`, 5 bugs are found at `Read Committed`, 2 bugs are found at `Read Uncommitted`, and 1 bug is found at `Serializable`. Among the 12 unique bugs, 10 bugs are isolation bugs. Note that, there is an overlap between the bugs that can be triggered at different isolation levels.

TABLE IV
BUGS REPORTED BY TROC

| DBMS | Isolation level | | | | Total | New | Duplicate |
|---|---|---|---|---|---|---|---|
| | RU | RC | RR | SER | (Unique) | (iBug) | (iBug) |
| MySQL | 1 | 1 | 3 | 0 | 5 (4) | 1 (1) | 3 (3) |
| MariaDB | 1 | 2 | 2 | 1 | 6 (4) | 3 (3) | 1 (1) |
| TiDB | - | 2 | 4 | - | 6 (4) | 3 (1) | 1 (1) |
| **Total** | 2 | 5 | 9 | 1 | 17 (12) | 7 (5) | 5 (5) |

TABLE V
NEW BUGS REPORTED BY TROC

| Issue | Isolation | iBug | Status | Severity |
|---|---|---|---|---|
| MySQL#104833 | RU, RC | Y | Verified | Serious |
| MariaDB#27992 | RC, SER | Y | Fixed | Critical |
| MariaDB#26643 | RU, RC | Y | Verified | Critical |
| MariaDB#26642 | RR | Y | Verified | Critical |
| TiDB#28212 | RR | Y | Verified | Moderate |
| TiDB#28092 | RC, RR | N | Verified | Moderate |
| TiDB#28095 | RC, RR | N | Verified | Minor |

The remaining 2 bugs can be triggered by using only one transaction, no matter what isolation level is used. We name them as transaction bugs. Listing 4 shows such a transaction bug that we find.

For the 12 unique bugs, 7 bugs (5 isolation bugs and 2 transaction bugs) have been verified as new bugs by developers (New), and one new isolation bug in MariaDB has been fixed by developers. The remaining 5 isolation bugs are marked as duplicate (Duplicate).

We list all our found new bugs in Table V. For these 7 new bugs, 4 bugs leave the database into incorrect states, and 3 bugs return incorrect query results. 1 isolation bug in MySQL is classified as *serious*, 3 isolation bugs in MariaDB are classified as *critical*, 1 isolation bugs in TiDB is classified as *moderate*, and the remaining 2 transaction bugs are classified as *moderate* and *minor*, respectively.

**Comparison.** We further investigate whether these 12 unique bugs can be revealed by existing approaches, e.g., SQLsmith [24], SQLancer [26]–[28], Elle [23] and Cobra [22]. These approaches cannot generate the bug-revealing data structures and SQL queries, or construct the corresponding transaction oracle. Thus, none of these bugs can be detected by these approaches theoretically. Specifically, existing verification approaches on isolation bugs, e.g., Elle [23] and Cobra [22], can only work on a simple *key-value* data model, which accesses data with $read(key)$ and $write(key)$. They cannot detect any of the 10 isolation bugs reported by Troc, which involve complex relational data models and SQL queries, e.g., writing multiple rows in the SQL statement $s22$ of Fig. 1. Existing testing techniques for DBMSs, e.g., SQLsmith [24] and SQLancer [26]–[28], cannot generate transaction test cases, and do not have a test oracle for transaction test cases.

We also investigate whether the 6 isolation bugs (which contain complete test cases) reported by Elle [23] and Cobra [22] can potentially be detected by Troc, i.e., whether their test cases violate Troc's oracle. Troc can detect 5 of these 6 isolation bugs. The remaining one isolation bug that Troc cannot detect requires more than two transactions to trigger, which is not supported by Troc now.

**Developer feedbacks.** For the 7 newly detected bugs, developers have fixed 1 isolation bug in MariaDB [54]. The remaining 6 bugs have not been fixed yet. Based on the feedbacks from developers, these 6 bugs can be divided into three categories. First, the developers think they are too hard to diagnose and fix (2/6). For example, the developer explains in MySQL#104833 [29] when we discuss the root cause with them: *"The exact cause of the bug is not found yet, but we are working on it very intensively .....".* Second, due to compatibility issues, the developers have not decided to fix the long-time lurking bugs for now, although the bugs have caused severe impact, e.g., incorrect query results (2/6). A developer mentions in MariaDB#26642 [55] *"This is the impact of a design decision of the InnoDB Repeatable Read, probably present since the very first release (MySQL 3.23.49). If we changed this now after all these years, some applications could be broken".* Third, developers just leave issues open for no reason (2/6). All these bugs belong to TiDB. Considering that it is a new rising DBMS, we think developers are possibly too busy to deal with a complex fix for our reported bugs.

## C. New Bugs

We explain our newly-found bugs under the pessimistic transaction model in detail. We have explained isolation bug **MySQL#104833** [29] found by Troc in Fig. 1. The test case in Fig. 1 also causes isolation bug **MariaDB#26643** [56]. We will not explain these two isolation bugs further.

**MariaDB#26642** [55]. Listing 2 shows an isolation bug at `Repeatable Read` in MariaDB. In the test case, two transactions $tx1$ and $tx2$ concurrently modify a table [(0, 0), (1, 1)]. $tx2$ firstly updates $c1$ in the second row as 10 (Line 6) and commits successfully. Then, $tx1$ queries the table by using *snapshot read* in Line 8 and obtains the correct result [(0, 0), (1, 1)]. $tx1$ further updates $c1$ in all rows as 10 later (Line 9). Since $tx2$ has committed, Line 9 can successfully update all rows in table $t$. For the third `SELECT` statement of $tx1$ (Line 10), $tx1$ should return all its modified data by Line 9, i.e., [(10, 0), (10, 1)], but its actual execution result is [(10, 0), (1, 1)]. Therefore, $tx1$ fails to see the effect of its own write in Line 9.

```
 1. /*init*/ CREATE TABLE t(c1 INT, c2 INT);
 2. /*init*/ INSERT INTO t VALUES(0,0),(1,1);
 3. /*tx1*/ BEGIN;
 4. /*tx1*/ SELECT * FROM t; -- [(0,0),(1,1)]
 5. /*tx2*/ BEGIN;
 6. /*tx2*/ UPDATE t SET c1 = 10 WHERE c2 = 1;
 7. /*tx2*/ COMMIT;
 8. /*tx1*/ SELECT * FROM t; -- [(0,0),(1,1)]
 9. /*tx1*/ UPDATE t SET c1 = 10 WHERE TRUE;
10. /*tx1*/ SELECT * FROM t; -- [(10,0),(1,1)]
11. /*tx1*/ COMMIT;
```

Listing 2. MariaDB#26642 reported at `Repeatable Read`

Interestingly, when diagnosing this isolation bug, we find it is **data-dependent**. We observe that, only when Line 6 and Line 9 update the same rows with the same data, this bug will occur. Otherwise, Line 10 can return the correct result. For example, if Line 6 is replaced by `UPDATE t SET c1 = 9 WHERE c2 = 1`, in which the original data 10 is changed to 9, Line 10 will correctly return [(10, 0), (10, 1)].

This isolation bug exists in MariaDB for a long time. We can trigger it at versions from 5.5 to the latest 10.7. The developers mark this bug as *critical*, but they have not decided to fix it yet, since *some applications (based on MariaDB) could be broken if the wrong behavior is fixed.*

Listing 2 can also trigger an isolation bug in TiDB for all public versions (**TiDB#28212** [57]). TiDB's developers explain its root cause. In TiDB, `SELECT` statements use *snapshot read* to read the data in its snapshot, while `UPDATE` statements use *current read* to read the latest rows and further add changed data to the transaction's change buffer. In Listing 2, $tx1$ first takes a snapshot of the initial table. After $tx2$ updates the table as [(0, 0), (10, 1)] in Line 6, $tx1$'s `SELECT` in Line 8 directly reads from its snapshot as [(0, 0), (1, 1)]. However, $tx1$'s `UPDATE` in Line 9 reads the latest data written by $tx2$, i.e., [(0, 0), (10, 1)], and further updates it as [(10, 0), (10, 1)]. Note that, the second row $(10, 1)$ is not changed by $tx1$, so it is not added into $tx1$'s change buffer. At last, the `SELECT` statement of $tx1$ in Line 10 reads from both its snapshot and change buffer, and keeps the second row as its initial snapshot version $(1, 1)$, thus wrongly returning [(10, 0), (1, 1)].

**MariaDB#27992** [54]. Listing 3 shows an isolation bug in MariaDB, which can occur at `Read Committed` and `Serializable` isolation levels. In this test case, $tx1$ first updates $c1$ as 5 in Line 5, and $tx2$ tries to delete all rows in Line 6, and is blocked since $tx2$ conflicts with $tx1$. Then, $tx1$ further updates $c1$ as 3, and commits. Now, the `DELETE` statement of $tx2$ in Line 6 is unblocked and further executed. Since the `DELETE` statement of $tx2$ in Line 6 deletes all rows in table $t$, the `SELECT` statement of $tx2$ in Line 9 should return an empty result. However, it returns [(3)] wrongly. Worse, table $t$ still contains value 3 after $tx2$ commits in Line 10. MariaDB developers have fixed this bug.

```
 1. /*init*/ CREATE TABLE t(c1 INT PRIMARY KEY);
 2. /*init*/ INSERT INTO t(c1) VALUES (8);
 3. /*tx1*/ BEGIN;
 4. /*tx2*/ BEGIN;
 5. /*tx1*/ UPDATE t SET c1 = 5;
 6. /*tx2*/ DELETE FROM t; -- tx2 is blocked
 7. /*tx1*/ UPDATE t SET c1 = 3;
 8. /*tx1*/ COMMIT; -- tx2 is unblocked
 9. /*tx2*/ SELECT * FROM t FOR UPDATE; -- [(3)]
10. /*tx2*/ COMMIT;
```

Listing 3. MariaDB#27992 reported at `Serializable`

**TiDB#28092** [58]. Listing 4 shows a transaction bug in TiDB. Unlike isolation bugs that require multiple transactions at a given isolation level, only one transaction in the test case can trigger this bug. The initial table is special. We restrict its first column as `NOT NULL`, but insert a `NULL` value with `IGNORE` option (Line 2). Transaction $tx1$ executes two `UPDATE` statements, i.e., $u1$ in Line 5 and $u2$ in Line 6. In this case, $u1$ can be successfully executed, but $u2$ throws an **unexpected error** *"Truncated incorrect DOUBLE value: ‘’"*. If we sequentially execute $u1$ and $u2$ in two different transactions, e.g., $tx2$ and $tx3$, respectively, both $u1$ and $u2$ can be successfully executed, without throwing any exception.

```
1. /*init*/ CREATE TABLE t(c1 BLOB NOT NULL, c2
   TEXT);
2. /*init*/ INSERT IGNORE INTO t VALUES(NULL,
   NULL);
3. /*init*/ INSERT INTO t VALUES(0x32,'aaa');
4. /*tx1*/ BEGIN;
5. /*tx1*/ UPDATE t SET c2 = 'abc'; -- [ ]
6. /*tx1*/ UPDATE t SET c2 = 'xyz' WHERE c1; --
   [Truncated incorrect DOUBLE value: ''']
7. /*tx1*/ COMMIT;
```

Listing 4. TiDB#28092, an interesting transaction bug

Another bug **TiDB#28095** [59] is similar to TiDB#28092, but has completely different triggering conditions. Although these two bugs are not isolation bugs according to their root causes, we can see that, Troc is capable to find some transaction bugs, in which a statement has different behavior when being executed independently and in a specific transaction.

## D. Other Experimental Statistics

**Test effort.** In our experiment, on average, Troc takes 0.3 seconds to generate a transaction test case (including database generation), and takes 5.3 seconds for the oracle to decide the outcome for a single transaction test case (including transaction execution). In total, it takes 200 to 60000 test runs (about 0.5 to 200 hours) for Troc to report a bug. Note that, this is not the number of test runs that can reveal a new bug, Troc may generate reports for the same bugs. If a bug has not been repaired by developers, Troc can potentially trigger the same bug with different test cases.

In addition, Troc discards about 12% of all test runs when it cannot decide whether there is an isolation bug (Section III-F).

**Code coverage.** To estimate how much code of the DBMSs we can test, we instrument MySQL and MariaDB[1], and run Troc on them for 24 hours. Troc achieves 20.6% and 25.1% line coverage for MySQL and MariaDB, respectively. The coverage appears to be low. However, this is expected, because Troc currently only focuses on transaction implementations. DBMSs also provide other features that we do not test, e.g., user management, configuration, and replication.

## V. Discussion

**False positives.** In Troc, we assume that the execution of a single SQL statement in DBMSs is always correct. In fact, the single SQL statement execution can be incorrect due to various reasons, e.g., logic bugs [25]–[28]. Therefore, our transaction oracle construction may obtain wrong oracle, and potentially introduce false positives. However, we have not observed such false positives in our experiment yet.

[1]We do not perform coverage experiment on TiDB, since we cannot find a proper coverage measure tool for Golang and Rust, which are used in TiDB.

**Migrating to new DBMSs.** In Troc, SQL and database generation is DBMS-related, since DBMSs usually have different dialects. However, Troc can utilize existing SQL and database generation approaches, e.g., SQLancer [49]. Therefore, Troc can be migrated to new DBMSs with little effort. For example, migrating Troc from MySQL to TiDB only introduces about 100 LOC changes, which are mainly used to access DBMS-related database structures, e.g., indexes.

**Supporting new isolation levels.** Troc has supported 4 isolation levels that are implemented in our target DBMSs. There are other isolation levels, e.g., `Read Stability` in DB2 [60]. To support other isolation levels, we need to adjust data visibility strategies (Table II) and lock strategies (Table III) according to new isolation levels. Troc's view construction and conflict analysis can be directly applied on these adjusted data visibility strategies and lock strategies.

**Supporting optimistic transactions.** We mainly focus on the pessimistic transaction model. Some DBMSs, e.g., TiDB and PostgreSQL, support the optimistic transaction model. To support the optimistic transaction model, we need to add two extra rules for DBMSs' behaviors. First, in spite of statement conflicts, all statements should not be blocked, but the results of conflict analysis are still stored. Second, when a transaction $tx1$ is to be committed, and it conflicts with a committed transaction $tx2$, then we use the database change history update rule for `ROLLBACK` statement (Section III-E4) to abort $tx1$. Based on these rules, Troc can construct oracles for transactions under the optimistic transaction model.

**Supporting more concurrent transactions.** We mainly focus on the scenarios where only two transactions submit their statements concurrently. Extending Troc to more than two transactions can introduce indeterminacy to transaction oracle construction. For example, given three transactions $tx1$, $tx2$ and $tx3$, both $tx1$ and $tx2$ are blocked by $tx3$. When $tx3$ is committed, which transaction is the first to be resumed is uncertain. A possible solution is to enumerate all scenarios and take all analysis results as the oracle.

## VI. Related Work

**Database testing.** Many approaches have been proposed for DBMS testing and graph database system testing [24]–[28], [48], [61]–[74]. SQLsmith [24] randomly generates SQL statements to detect crash bugs in DBMSs. Squirrel [48] performs query generation guided by code coverage to test DBMSs. LEGO [71] generates SQL sequences with abundant types to improve DBMS fuzzing coverage. DynSQL [72] utilizes stateful fuzzing to test DBMSs and find deep bugs. Rigger et al. presents some approaches to construct oracles for `SELECT` statements, e.g., PQS [26], TLP [27] and NoREC [28]. QPG [69] utilizes query plans to guide database state mutation for detecting bugs. DQE [67] detects logic bugs by differentially executing `SELECT`, `UPDATE` and `DELETE` statements in DBMSs. APOLLO [61] detects performance regression bugs by generating regression-triggering queries. Amoeba [63] detects performance bugs by generating two semantically equivalent queries and comparing their execution time. Some works utilize differential testing to effectively test DBMSs and graph database systems [25], [64], [65], [68], [74]. However, these approaches cannot construct test oracle for detecting isolation bugs in DBMSs.

**Transaction verification.** Biswas et al. [21] utilizes an axiomatic framework to characterize whether a transaction execution history satisfies a certain isolation level. Cobra [22] verifies the serializability of key-value stores. To improve scalability, a SMT solver is used to tackle the computational explosion problem. Elle [23] finds isolation bugs by checking the transaction execution histories on specific designed consistency models (e.g., AppendList). These works mainly rely on transaction execution histories of specific database models (e.g., $key-value$ data model), and cannot test many transaction features in modern DBMSs, e.g., reading multiple rows in a SQL statement. In contract, Troc supports generating transactions with complex relational data models and features and detects isolation bugs by oracle construction for a pair of concurrent transactions.

**Combating transaction concurrency problems in database applications.** In recent years, researchers have presented several approaches to detect or debug transaction concurrency problems for applications relying on DBMSs [75]–[81]. Tang et al. [81] conducts a comprehensive study on transaction concurrency problems in ad hoc transactions among database applications. Brutschy et al. [77] applies static analysis to detect serializability violation behaviors in database applications. Deng et al. [75] and Luo et al. [76] propose several approaches for detecting transaction concurrency problems in database applications. CLOTHO [78], IsoDiff [79] and MonkeyDB [80] detect transaction isolation violations in database applications. These works aim to combat transaction concurrency problems in database applications, which are orthogonal to Troc.

## VII. Conclusion

DBMSs can violate their claimed transaction isolation levels and introduce isolation bugs. In this paper, we propose a novel, general and effective transaction testing approach, Troc, to automatically detect isolation bugs in DBMSs. The core idea of Troc is to construct the oracle for each statement in two concurrent transactions with the guidance of isolation levels. We have applied Troc on three widely-used DBMSs, i.e., MySQL, MariaDB and TiDB, and found 5 previously-unknown isolation bugs in them. In the future, we plan to extend Troc to test more complex transaction execution scenarios, e.g., optimistic transactions and more than two concurrent transactions.

## 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.

## References

[1] (2022) MySQL. [Online]. Available: https://www.mysql.com

[2] (2022) MariaDB. [Online]. Available: https://mariadb.org

[3] (2022) PostgreSQL. [Online]. Available: https://www.postgresql.org

[4] (2022) TiDB, PingCAP. [Online]. Available: https://pingcap.com

[5] (2022) CockroachDB. [Online]. Available: https://www.cockroachlabs.com/

[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 (SIGFIDET)*, 1974, pp. 249–264.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[8] P. M. Lewis, A. Bernstein, and M. Kifer, "Databases and transaction processing: An application-oriented approach," *ACM SIGMOD Record*, vol. 31, no. 1, pp. 74–75, 2002.

[9] (2022) The ANSI isolation levels. [Online]. Available: http://www.adp-gmbh.ch/ora/misc/isolation_level.html

[10] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1995, pp. 1–10.

[11] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proceedings of the VLDB Endowment (VLDB)*, vol. 7, no. 3, pp. 181–192, 2013.

[12] D. Potier and P. Leblanc, "Analysis of locking policies in database management systems," *Communications of the ACM*, vol. 23, no. 10, pp. 584–593, 1980.

[13] S. Sippu and E. Soisalon-Soininen, *Lock-Based Concurrency Control*, 2014, pp. 125–158.

[14] D. P. Reed, "Naming and synchronization in a decentralized computer system," Tech. Rep., 1978.

[15] P. A. Bernstein and N. Goodman, "Multiversion concurrency control—theory and algorithms," *ACM Transactions on Database Systems (TODS)*, vol. 8, no. 4, pp. 465–483, 1983.

[16] M. J. Carey, "Improving the performance of an optimistic concurrency control algorithm through timestamps and versions," *IEEE Transactions on Software Engineering (TSE)*, vol. SE-13, no. 6, pp. 746–751, 1987.

[17] X. Song and J. W.-S. Liu, "Performance of multiversion concurrency control algorithms in maintaining temporal consistency," in *Proceedings of Annual International Computer Software and Applications Conference (COMPSAC)*, 1990, pp. 132–139.

[18] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[19] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, "TicToc: Time traveling optimistic concurrency control," in *Proceedings of International Conference on Management of Data (SIGMOD)*, 2016, pp. 1629–1642.

[20] (2022) Hermitage. [Online]. Available: https://github.com/ept/hermitage

[21] R. Biswas and C. Enea, "On the complexity of checking transactional consistency," in *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2019, pp. 165:1–165:28.

[22] C. Tan, C. Zhao, S. Mu, and M. Walfish, "Cobra: Making transactional key-value stores verifiably serializable," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 63–80.

[23] K. Kingsbury and P. Alvaro, "Elle: Inferring isolation anomalies from experimental observations," *Proceedings of the VLDB Endowment (VLDB)*, vol. 14, no. 3, pp. 268–280, 2020.

[24] (2022) SQLsmith. [Online]. Available: https://github.com/anse1/sqlsmith

[25] D. R. Slutz, "Massive stochastic testing of SQL," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 1998, pp. 618–622.

[26] 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.

[27] ——, "Finding bugs in database systems via query partitioning," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 211:1–211:30, 2020.

[28] ——, "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.

[29] (2022) Inconsistent behaviors of UPDATE under Read Uncommitted. [Online]. Available: https://bugs.mysql.com/bug.php?id=104833

[30] 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.

[31] (2022) DB-Engines. [Online]. Available: https://db-engines.com/en/ranking

[32] (2022) GitHub. [Online]. Available: https://github.com/

[33] A. Adya, "Weak consistency: A generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[34] A. Adya, B. Liskov, and P. O'Neil, "Generalized isolation level definitions," in *Proceedings of International Conference on Data Engineering (ICDE)*, 2000, pp. 67–78.

[35] A. Cerone, G. Bernardi, and A. Gotsman, "A framework for transactional consistency models with atomic visibility," in *Proceedings of International Conference on Concurrency Theory (CONCUR)*, 2015, pp. 58–71.

[36] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Scalable atomic visibility with RAMP transactions," *ACM Transactions on Database Systems (TODS)*, vol. 41, no. 3, pp. 15:1–15:45, 2016.

[37] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev, "Serializability for eventual consistency: Criterion, analysis, and applications," in *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017, pp. 458–472.

[38] (2022) MySQL isolation. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html

[39] (2022) Isolation levels in MariaDB. [Online]. Available: https://mariadb.com/kb/en/set-transaction/

[40] (2022) TiDB Isolation. [Online]. Available: https://docs.pingcap.com/tidb/v5.0/transaction-isolation-levels

[41] 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.

[42] N. Bruno and S. Chaudhuri, "Flexible database generators," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2005, pp. 1097–1107.

[43] E. F. Codd, "Relational completeness of data base sublanguages," *Research Report*, 1972.

[44] 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.

[45] 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.

[46] S. A. 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.

[47] (2022) go-randgen. [Online]. Available: https://github.com/pingcap/go-randgen

[48] 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.

[49] (2022) SQLancer. [Online]. Available: https://www.manuelrigger.at/dbms-bugs/

[50] (2022) InnoDB transaction model. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-model.html

[51] J. N. Gray, R. A. Lorie, and G. R. Putzolu, "Granularity of locks in a shared data base," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 1975, pp. 428–451.

[52] (2022) Next-key locking in MySQL. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/innodb-next-key-locking.html

[53] (2022) Execution plan information in MySQL. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/execution-plan-information.html

[54] (2022) DELETE fails to delete record after blocking is released. [Online]. Available: https://jira.mariadb.org/browse/MDEV-27992

[55] (2022) Weird SELECT view when a record is modified to a same value by two transactions. [Online]. Available: https://jira.mariadb.org/browse/MDEV-26642

[56] (2022) Inconsistent behaviors of UPDATE under RU & RC isolation level. [Online]. Available: https://jira.mariadb.org/browse/MDEV-26643

[57] (2022) Weird SELECT view when a record is modified to the same value by two transactions. [Online]. Available: https://github.com/pingcap/tidb/issues/28212

[58] (2022) UPDATE has inconsistent behaviors in a transaction. [Online]. Available: https://github.com/pingcap/tidb/issues/28092

[59] (2022) UPDATE with CAST has inconsistent behaviors in transaction. [Online]. Available: https://github.com/pingcap/tidb/issues/28095

[60] (2022) Isolation levles - IBM documentation. [Online]. Available: https://www.ibm.com/docs/en/db2/10.5?topic=issues-isolation-levels

[61] 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.

[62] 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 International Conference on Software Engineering: Software Engineering in Practice (ICSE SEIP)*, 2021, pp. 328–337.

[63] X. Liu, Q. Zhou, J. Arulraj, and A. Orso, "Automatic detection of performance bugs in database systems using equivalent queries," in *Proceedings of IEEE/ACM SIGSOFT International Conference on Software Engineering (ICSE)*, 2022, pp. 225–236.

[64] 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.

[65] 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, pp. 35:1–35:12.

[66] 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.

[67] J. Song, W. Dou, Z. Cui, Q. Dai, W. Wang, J. Wei, H. Zhong, and T. Huang, "Testing database systems via differential query execution," in *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023.

[68] R. Yang, Y. Zheng, L. Tang, W. Dou, W. Wang, and J. Wei, "Randomized differential testing of RDF stores," in *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE Demo)*, 2023.

[69] J. Ba and M. Rigger, "Testing database engines via query plan guidance," in *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023.

[70] 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.

[71] J. Liang, Y. Chen, Z. Wu, J. Fu, M. Wang, Y. Jiang, X. Huang, T. Chen, J. Wang, and J. Li, "Sequence-oriented DBMS fuzzing," in *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2023.

[72] Z.-M. Jiang, J.-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.

[73] Z. Hua, W. Lin, L. Ren, Z. Li, L. Zhang, W. Jiao, and T. Xie, "GDsmith: Detecting bugs in Cypher graph database engines," in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023.

[74] W. Lin, Z. Hua, L. Zhang, and T. Xie, "GDiff: Automated differential performance testing for graph database systems," in *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023.

[75] Y. Deng, P. Frankl, and Z. Chen, "Testing database transaction concurrency," in *Proceedings of IEEE International Conference on Automated Software Engineering (ASE)*, 2003, pp. 184–193.

[76] H. Luo, M. Masud, and H. Ural, "Detecting offline transaction concurrency problems," *Journal of Software*, vol. 7, pp. 1855–1860, 2012.

[77] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev, "Static serializability analysis for causal consistency," in *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018, pp. 90–104.

[78] K. Rahmani, K. Nagar, B. Delaware, and S. Jagannathan, "CLOTHO: Directed test generation for weakly consistent database systems," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 117:1–117:28, 2019.

[79] Y. Gan, X. Ren, D. Ripberger, S. Blanas, and Y. Wang, "IsoDiff: Debugging anomalies caused by weak isolation," *Proceedings of the VLDB Endowment (VLDB)*, vol. 13, no. 12, pp. 2773–2786, 2020.

[80] R. Biswas, D. Kakwani, J. Vedurada, C. Enea, and A. Lal, "MonkeyDB: Effectively testing correctness under weak isolation levels," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 132:1–132:27, 2021.

[81] C. Tang, Z. Wang, X. Zhang, Q. Yu, B. Zang, H. Guan, and H. Chen, "Ad hoc transactions in web applications: The good, the bad, and the ugly," in *Proceedings of International Conference on Management of Data (SIGMOD)*, 2022, pp. 4–18.