Differentially Testing Database Transactions for Fun and Profit

Ziyu Cui
State Key Lab of Computer Science at ISCAS, University of CAS
China
cuiziyu20@otcaix.iscas.ac.cn

Wensheng Dou*
State Key Lab of Computer Science at ISCAS, University of CAS, University of CAS Nanjing College
China
wsdou@otcaix.iscas.ac.cn

Qianwang Dai
State Key Lab of Computer Science at ISCAS, University of CAS
China
daiqianwang19@otcaix.iscas.ac.cn

Jiansen Song
State Key Lab of Computer Science at ISCAS, University of CAS
China
songjiansen20@otcaix.iscas.ac.cn

Wei Wang
Jun Wei*
State Key Lab of Computer Science at ISCAS, University of CAS, Nanjing Institute of Software Technology
China
{wangwei,wj}@otcaix.iscas.ac.cn

Dan Ye
State Key Lab of Computer Science at ISCAS, University of CAS
China
yedan@otcaix.iscas.ac.cn

ABSTRACT

Database Management Systems (DBMSs) utilize transactions to ensure the consistency and integrity of data. Incorrect transaction implementations in DBMSs can lead to severe consequences, e.g., incorrect database states and query results. Therefore, it is critical to ensure the reliability of transaction implementations.

In this paper, we propose DT^2, an approach for automatically testing transaction implementations in DBMSs. We first randomly generate a database and a group of concurrent transactions operating the database, which can support complex features in DBMSs, e.g., various database schemas and cross-table queries. We then leverage differential testing to compare transaction execution results on multiple DBMSs to find discrepancies. The non-determinism of concurrent transactions can affect the effectiveness of our method. Therefore, we propose a transaction test protocol to ensure the deterministic execution of concurrent transactions.

We evaluate DT^2 on three widely-used MySQL-compatible DBMSs: MySQL, MariaDB and TiDB. In total, we have detected 10 unique transaction bugs and 88 transaction-related compatibility issues from the observed discrepancies. Our empirical study on these compatibility issues shows that DBMSs suffer from various transaction-related compatibility issues, although they claim that they are compatible. These compatibility issues can also lead to serious consequences, e.g., inconsistent database states among DBMSs.

*Wensheng Dou and Jun Wei are the corresponding authors. CAS is the abbreviation of Chinese Academy of Sciences. ISCAS is the abbreviation of Institute of Software, Chinese Academy of Sciences.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556924

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **General and reference** → *Empirical studies.*

KEYWORDS

Database transaction, differential testing, isolation level, compatibility issue

ACM Reference Format:
Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. 2022. Differentially Testing Database Transactions for Fun and Profit. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556924

1 INTRODUCTION

Database Management Systems (DBMSs) play an essential role in storing and retrieving data. Specially, relational DBMSs, e.g., MySQL [13], PostgreSQL [14], SQLite [17], and TiDB [21], utilize Structured Query Language (SQL) [43] to create, modify and query data, and have been widely used in many applications, e.g., e-commerce applications and mobile applications.

Relational DBMSs utilize transactions to guarantee data consistency and integrity. Transactions have ACID properties, i.e., Atomicity, Consistency, Isolation, and Durability. A transaction is a logical unit that consists of a group of SQL statements to retrieve and manipulate data. DBMSs execute these SQL statements atomically, and ensure that they are executed as a whole in spite of concurrency and system failures.

In DBMSs, multiple transactions should be executed in isolation from each other. However, if a DBMS adopts a stronger isolation level, its performance will degrade more. Therefore, academia and industry have proposed dozens of isolation levels [1, 28–31, 34, 41]. DBMSs usually provide several common isolation levels for developers, e.g., `Read Uncommitted`, `Read Committed`, `Repeatable Read`, and `Serializable` in MySQL [22] and MariaDB [15].

DBMSs usually adopt complex mechanisms to support transactions, e.g., multi-version concurrency control [35, 57, 62], and optimistic concurrency control [54, 66]. On the one hand, buggy

transaction implementations in DBMSs, i.e., transactions bugs, can violate their claimed ACID properties. Transaction bugs can lead to serious consequences, e.g., incorrect query results and database states. On the other hand, some DBMSs claim that they are compatible, but their transaction implementations suffer from subtle discrepancies for the same isolation level, e.g., compatibility issues. However, these compatibility issues are not well documented and studied, and can introduce challenges for database migration.

To improve the correctness of transaction implementations, existing approaches mainly focus on verifying whether DBMSs provide the transaction isolation guarantees that they claim [37, 41, 42, 53, 64]. These approaches utilize simplified database structure (e.g., $key - value$ model in Elle [53] and Cobra [64]), and analyze the read / write history of *one row* indexed by a *key* to find transaction bugs. However, relational DBMSs usually support many complex features, e.g., data structures (e.g., data types and indexes), and SQL statements (e.g., cross-table queries). Therefore, existing approaches cannot detect transaction bugs that utilize these features. Some approaches, e.g., PQS [60], NoREC [58], TLP [59] and RAGS [61], can detect DBMS bugs that involve these complex features in *single* `SELECT` statements. However, they cannot detect transaction bugs that involve multiple SQL statements and isolation levels.

To effectively detect transaction bugs and compatibility issues in DBMSs, we propose a general transaction testing approach, *Differentially Testing Database Transactions* (*DT*^{2}). We leverage differential testing to build a test oracle for transaction implementations in DBMSs. Specially, we first generate a random database, and then generate a group of concurrent transactions that interact with the database. We then execute these transactions on DBMSs under test, and compare the transaction execution results to find discrepancies. To leverage differential testing for database transaction testing, we need to address the following technical issues.

- **Different SQL execution semantics.** Relational DBMSs usually support standardized SQL to manipulate data. However, DBMSs may exhibit different SQL execution semantics. For example, MySQL and PostgreSQL adopt different mechanisms for data type constraints and implicit data type conversions. This can greatly affect the effectiveness of differential testing. We observe that some DBMSs claim that they are designed to be compatible with other popular DBMSs, and adopt the same SQL execution semantics. For example, MariaDB [12] is forked from MySQL and maintains a high level of compatibility with MySQL, and TiDB is claimed to be compatible with MySQL, and CockroachDB [2] is compatible with PostgreSQL. Therefore, we choose compatible DBMSs as target DBMSs, e.g., MySQL-compatible DBMSs (MySQL, MariaDB, TiDB, etc).
- **SQL dialects.** Although compatible DBMSs support common SQL execution semantics, each DBMS can provide some extra extensions, i.e., dialects. For example, TiDB does not support the `SPATIAL` data type in MySQL [3–5]. To handle this issue, we generate transaction test cases that only involve common SQL features that all DBMSs support, e.g., `JOIN`, `GROUP BY`, and rich `WHERE` expressions.
- **Non-determinism transaction execution.** Given a group of concurrent transactions, their execution will be affected by non-deterministic schedule and transaction modes in a DBMS, and their execution results will be non-deterministic. To handle this issue, we propose a transaction test protocol under the pessimistic transaction mode, in which we can control the execution order of each statement in transactions, and can obtain deterministic transaction execution results.

To demonstrate the effectiveness of DT^2, we evaluate it on three widely-used MySQL-compatible DBMSs under their pessimistic transaction mode, i.e., MySQL, MariaDB, and TiDB. DT^2 has detected 146 discrepancies among the three DBMSs. From these discrepancies, we have identified 16 unique bugs, including 4 bugs in MySQL, 7 bugs in MariaDB, and 5 bugs in TiDB. Among these 16 bugs, 10 bugs are transaction-related bugs while the remaining 6 bugs are transaction-unrelated bugs (i.e., a single SQL statement can trigger a bug). We have submitted these bugs to developers, and they have confirmed 6 bugs as new bugs. These 6 new bugs have caused serious consequences, e.g., incorrect database states (2 bugs), incorrect query results (1 bug), missing expected errors (2 bugs), and reporting unexpected errors (1 bug). Our newly-detected transaction bugs arouse some interesting discussions. For example, in a MariaDB bug [7], developers state that *"For fixing this bug, I can offer a wild idea that I do not think can be implemented easily"*. Since there is no better solution at the moment, developers reverted an earlier fix to prevent this bug.

From the remaining discrepancies among the three DBMSs, we reveal 88 compatibility issues among these DBMSs' transaction implementations. These compatibility issues are caused by different design choices of these DBMSs. We further conduct an empirical study on these observed compatibility issues. For each compatibility issue, we analyze and summarize its triggering scenario, root cause, and consequence. Our study shows that these MySQL-compatible DBMSs suffer from various compatibility issues on transaction implementations, even though they claim they are compatible. For example, TiDB and MySQL create snapshots for `SELECT` statements at different locations under `Repeatable Read` isolation level. These compatible issues can cause inconsistencies among DBMSs, e.g., inconsistent database states, inconsistent query results, and inconsistent deadlock.

In summary, we make the following contributions.

- We propose the first differential testing approach for transaction implementations of DBMSs.
- We implement the approach as DT^2 and use it to test three widely-used MySQL-compatible DBMSs, i.e., MySQL, MariaDB and TiDB. DT^2 finds 10 unique transaction bugs and 88 transaction-related compatibility issues among these DBMSs. DT^2 is publicly available at https://github.com/tcse-iscas/DT2.
- We conduct the first empirical study on transaction-related compatibility issues in MySQL, MariaDB and TiDB, and reveal their triggering scenarios, root causes, and consequences. We hope our study can shed light on the transaction behavior specification, and facilitate database migration among DBMSs.

2 PRELIMINARIES

2.1 Relational DBMSs and SQL

Relational DBMSs, e.g., MySQL [13], PostgreSQL [14], SQLite [17] and TiDB [21], are widely used in many applications, e.g., e-commerce applications and mobile applications. Relational DBMSs organize

Table 1: Tested DBMSs in our evaluation.

DBMS	DB-Engines Ranking	Stars	Isolation Levels
MySQL	2	7.6K	RU, RC, RR, SER
MariaDB	13	4.2K	RU, RC, RR, SER
TiDB	112	30.9K	RC, RR

data based on the relational data model proposed by Codd [47] and store data as tables. A database can contain multiple tables. Users utilize Structured Query Language (SQL) [43] to interact with DBMSs, and perform data query, insertion, deletion, and modification.

MySQL [13] and PostgreSQL [14] are two mature DBMSs with well transaction support. As they are widely-used open-source DBMSs, many DBMSs claim to be compatible with them and can be classified into MySQL-compatible and PostgreSQL-compatible, respectively. For example, MariaDB is compatible with MySQL, while CockroachDB is a PostgreSQL-compatible commercial DBMS. Although both MySQL and PostgreSQL provide SQL to retrieve data and have many things in common, there are some significant differences between them. For example, PostgreSQL offers more complex data types, and supports different implicit data type conversions. The data type constraints in PostgreSQL are stricter than that in MySQL.

2.2 Tested DBMSs

Since PostgreSQL and MySQL have largely different SQL execution semantics (e.g., different implicit data type conversions), it would involve large implementation efforts to identify discrepancies between PostgreSQL-compatible DBMSs and MySQL-compatible DBMSs. In our work, we choose MySQL-compatible DBMSs as our research targets. Note that, our approach can be potentially extended to test PostgreSQL-compatible DBMSs.

Based on the DB-Engines Ranking [6] and Github stars [9], we finally choose three widely-used MySQL-compatible DBMSs, i.e., MySQL, MariaDB and TiDB, as shown in Table 1. MySQL and MariaDB are traditional relational DBMSs, and TiDB is a NewSQL DBMS that combines the relational model with distributed support.

Transaction modes. DBMSs usually adopt two transaction execution modes, i.e., pessimistic and optimistic transaction modes. In the pessimistic transaction mode, a transaction needs to acquire locks on the accessed data. Then the later transactions will be blocked until their accessed data are unlocked. In the optimistic transaction mode, a transaction during running can access data without acquiring locks on them. Before committing, each transaction verifies whether other transactions have modified the data it has accessed. If yes, the committing transaction is rolled back and can be restarted.

MySQL and MariaDB only support the pessimistic transaction mode. TiDB supports both pessimistic and optimistic transaction modes [19, 20], and uses the pessimistic transaction mode by default. To perform differential testing on them, we test MySQL, MariaDB and TiDB under the pessimistic transaction mode, which all tested DBMSs support.

Isolation levels. In DBMSs, multiple transactions should be executed in isolation from each other. However, if a DBMS adopts a stronger isolation level among transactions, its performance will degrade more. To make a tradeoff between consistency and performance, DBMSs usually provide multiple isolation levels for developers [1, 28–31, 34, 41].

The isolation levels supported by our tested DBMSs are shown in Table 1. Basically, MySQL and MariaDB support four isolation levels, i.e., `Read Uncommitted`, `Read Committed`, `Repeatable Read`, and `Serializable` [1, 28, 34], while TiDB only supports two isolation levels, i.e., `Read Committed` and `Repeatable Read`. We test these DBMSs at all the four isolation levels and compare test results at the same isolation level.

Here we briefly explain these four isolation levels. Note that, all the four isolation levels prevent other transactions from overwriting data modified by uncommitted transactions.

- `Read Uncommitted (RU).` RU allows a transaction to read data updated by uncommitted transactions.
- `Read Committed (RC).` RC only allows a transaction to read data already committed by other transactions and their own modifications.
- `Repeatable Read (RR).` RR allows a transaction to read data committed by other transactions before the transaction started, as well as data modified by the transaction itself.
- `Serializable (SER).` SER is the strictest isolation level among the four isolation levels. At `Serializable`, the execution of concurrent transactions is equivalent to an execution in their certain sequential order.

3 MOTIVATING EXAMPLES

Transaction bug. Figure 1 shows a test case that triggers a real-world transaction bug in MariaDB [11] at `Repeatable Read` and `Serializable`. We find this bug by differentially testing MySQL and MariaDB, and developers have confirmed this transaction bug, and classified it as a `critical` bug.

The initial table contains one column $c1$ (primary key) and a record 3. Two transactions $t1$ and $t2$ are executed concurrently on this table. $t1$ first updates $c1$ to 5 (Line 4), and then $t2$ tries to delete all data in the table (Line 6), and is blocked under the pessimistic transaction mode. Then, $t1$ inserts a value 2 into $c1$ (Line 7). However, both MySQL and MariaDB encounter a conflict at the `INSERT` statement, resulting in a deadlock[1] that occurs in $t2$. In this case, $t2$ is rolled back. $t1$ then is committed (Line 8). We pass the remaining statements in $t2$ to DBMSs. $t2$ inserts a value 1 into $c1$ (Line 9) and executes `ROLLBACK` statement (Line 10). We finally retrieve the table, and observe that the results in two DBMSs are different. The database state in MySQL is {1, 2, 5}, while that in MariaDB is {2, 5}.

The root cause behind this bug is that the implementation of MariaDB for handling rolled back transactions in which a deadlock

[1] MySQL and MariaDB utilize gap lock at `Repeatable Read` and `Serializable` to prevent other transactions from inserting new records between two records. Because column $c1$ is a primary key, $t1$ executes `UPDATE` by deleting the original value 3 with a gap lock and an exclusive lock, and inserting the new value 5 (Line 4). $t2$ also requires a gap lock and an exclusive lock on value 3 for `DELETE` but sets these locks as waiting states because of lock conflict (Line 6). The `INSERT` operation in $t1$ (Line 7) tries to insert 2 but is blocked by the $t2$'s gap lock on value 3. Thus a deadlock occurs.

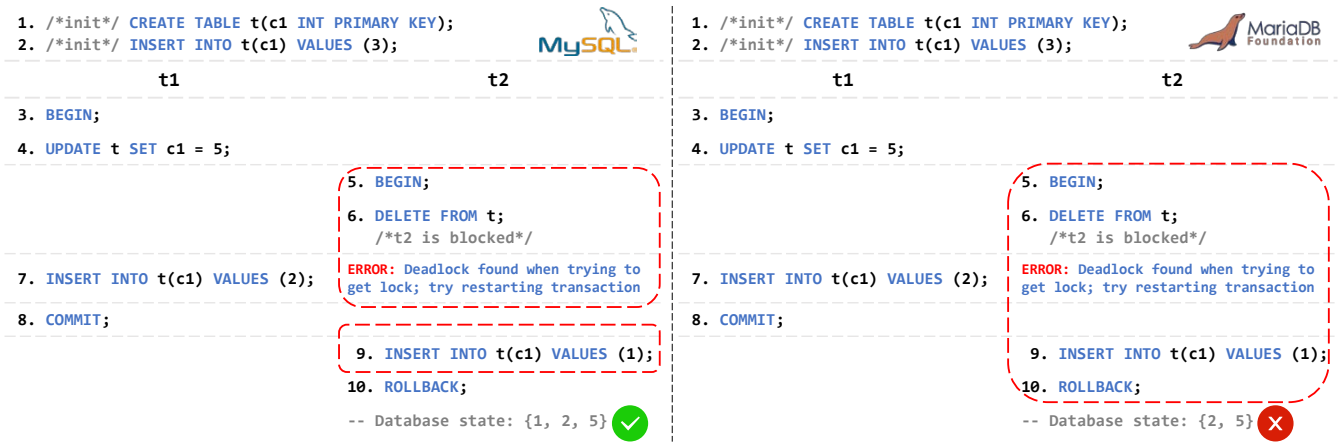

Figure 1: A motivating example that triggers a `critical` MariaDB transaction bug [11] at `Repeatable Read` and `Serializable` isolation levels under the pessimistic transaction mode. This bug leads to incorrect database state when a deadlock happens in transaction $t2$. The dotted rectangles show the real transaction scopes of transaction $t2$ in MySQL and MariaDB.

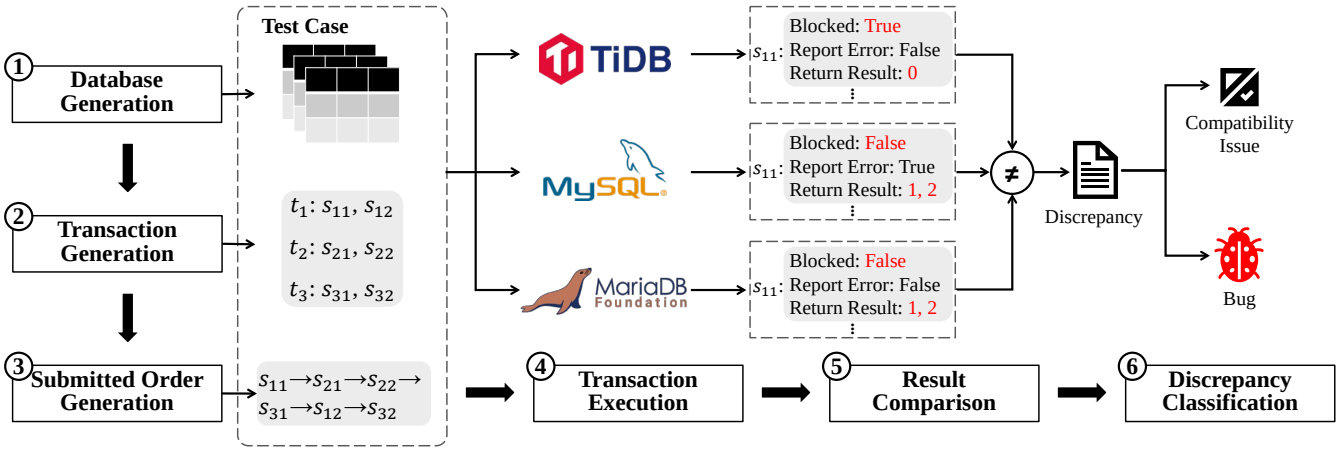

Figure 2: The workflow of DT^2.

occurs is problematic. In MySQL, if a transaction encounters a deadlock at statement $s1$, this transaction will be rolled back and terminated. The statements after $s1$ will be executed without the original transaction. As shown in Figure 1, the statements (Line 9-10) after the statement that reports the deadlock (Line 6) are treated as individual transactions, which are automatically committed after execution. Therefore, the `ROLLBACK` operation in Line 10 does not take effect.

However, MariaDB aborts a transaction that encounters a deadlock without terminating this transaction. Thus, subsequent statements are still executed within the original transaction. In this test case, $t2$ can roll back the operation in Line 9. MariaDB developers state that *"the error is that the second INSERT statement is being accepted for execution"*. Their advice on how to fix the bug is when $t2$ inserts the value 1, *"the SQL layer must return an error that the transaction was aborted"*.

Compatibility issue example. The test case above triggers a deadlock in MySQL, but it does not trigger a deadlock in MariaDB version 10.7.3 or later at `Repeatable Read` and `Serializable`. That is, $t2$ remains blocked after $t1$ executes `INSERT` statement in MariaDB (Line 7). After $t1$ is committed (Line 8), $t2$ is resumed, completes `DELETE` operation, and is committed. DT^2 reports this discrepancy between MySQL and MariaDB with inconsistent deadlock, and we recognize it as a compatibility issue.

The impact of this compatibility issue is serious. First, it is inconsistent whether a deadlock occurs. This test case cannot be completed due to the deadlock in MySQL, while it is executed successfully in MariaDB. Moreover, this issue leaves database into an inconsistent state. After a deadlock occurs in the transaction, the remaining statements are executed in individual transactions that contain only one statement. Thus, the `INSERT` statement in $t2$ cannot be rolled back, causing the database final state is {1, 2, 5} in

MySQL. However, MariaDB rolls back the INSERT statement in $t2$, so the database final state is {2, 5}. As existing DBMS differential testing approaches [50, 61, 63, 67] cannot perform on transactions, this compatibility issue cannot be detected.

4 APPROACH

We propose DT^2 to detect discrepancies in transaction execution behaviors among different DBMSs under the pessimistic transaction mode. DT^2 generates test cases, and performs differential testing that executes the same test cases on all target DBMSs. If the results of a test case executed by any two DBMSs are different, DT^2 generates a test report for subsequent analysis.

Figure 2 shows the overview of DT^2. First, we randomly create tables that contain some random data (①). Based on these tables, we generate several transactions with random SQL statements (②). We then generate a submitted order in which we expect the DBMS to execute the generated transactions concurrently (③). The generated tables, transactions, submitted order, and an isolation level that target DBMSs support form a test case. At the tested isolation level, we submit the statements in the the submitted order one by one to each DBMS for execution (④). Then, we compare the execution results of different DBMSs at the same isolation level, including whether the statement is blocked, whether the statement triggers a warning, error or deadlock, the results of query statements, and the database's final state (⑤). If the execution results are different, we generate a test report. In the last step, we manually simplify the test case in the test report, and analyze whether it is a bug. If not, the simplified test case is classified as a compatibility issue (⑥).

4.1 Transaction Test Case Generation

A test case consists of a database, a group of concurrent transactions, a submitted order and a tested isolation level. Three steps (i.e., ①, ②, and ③ in Figure 2) are required in randomly test case generation. Since SQL statements are the basis for generating test cases, we will describe SQL statement generation and the first two steps of our approach in details as follows. Note that, there have been a lot of approaches about random generation of databases [36, 39, 52] and SQL statements [18, 40, 50, 61]. We generate them mainly based on SQLancer [16], and slightly revise the approach for our target. We briefly describe random database and SQL statement generation for completeness.

Database Generation. As we focus on transactions, we create initial database using only the simplest table creating statements, i.e., CREATE and INSERT, of which CREATE statement includes CREATE TABLE and CREATE INDEX. We observe that discrepancies can usually be triggered on simple databases, so we build at most *maxTable* (3 by default) tables containing at most *maxColumn* (4 by default) for each table. To find the largest common subset of data types covered by DBMSs, we investigate and support all their data types. For example, since TiDB does not support Spatial data type, columns in the tables are randomly assigned data types except Spatial types, including Numeric, String, etc. To cover as many table structures as possible, we randomly add constraints on columns, e.g., PRIMARY KEY, UNIQUE, CHECK and NOT NULL, and column attributes like DEFAULT and AUTO_INCREMENT. Finally, we populate each table with at most 5 rows by executing randomly generated INSERT statements.

SQL statement generation. We generate SQL statements based on SQL syntax supported by tested DBMSs. SQL statements usually require some parameters. For example, the parameters of SELECT statements include the selected tables, columns and constants. We randomly select some generated tables and columns to populate the statement parameters. For constant parameters, we adopt two strategies. First, we randomly generate constant values. Second, we randomly pick the existing data in generated tables. We randomly choose one strategy at a time to generate constant values.

Differential testing should use the same semantic input, but different DBMSs have slightly different SQL dialects. For example, MySQL and MariaDB support SELECT FOR SHARE statements, while TiDB does not. Therefore, we support the largest common SQL subset supported by our tested DBMSs. We will not generate SQL statements that our tested DBMSs cannot execute. Even so, since we treat DBMSs as a black box, we can support testing many complex features of DBMSs with little effort. DT^2 can support SQL statements implemented by more than one DBMS, e.g., SELECT, SELECT FOR SHARE, SELECT FOR UPDATE, INSERT, UPDATE and DELETE, many SQL features, e.g., JOIN, GROUP BY, and rich expressions, e.g., various WHERE predicates and functions.

Transaction generation. Our transaction test case explicitly starts a transaction using a BEGIN statement and randomly ends it with a COMMIT statement (apply all changes in the transaction) or a ROLLBACK statement (roll back all changes in the transaction). Other generated SQL statement types include SELECT, SELECT FOR SHARE, SELECT FOR UPDATE, INSERT, UPDATE, and DELETE. A transaction that we generate consists of one to *maxStmt* SQL statements. In our experiment, we observe that using a small number of SQL statements can trigger discrepant transaction execution behaviors among DBMSs. Therefore, *maxStmt* is set to 7 by default.

4.2 Deterministic Transaction Execution

In differential testing, the same input is sent to the tested DBMSs for execution. Therefore, the execution orders of the statements in transactions, which are part of the testing inputs, need to be unified when DBMSs execute the same concurrent transactions. However, it is not easy to determine their execution orders. Generally, the submitted order of statements in transactions is enumerable. As such, if statements in transactions are executed one by one on a DBMS following certain submitted order, their execution order is determined by the transaction implementation and the given isolation level in the DBMS. Inspired by this observation, we propose a deterministic transaction test protocol, which submits transaction statements in our generated transaction test cases to a DBMS one by one in a randomly generated submitted order.

Submitted order generation. The submitted order indicates the initial order in which a DBMS accepts statements in concurrent transactions. Concurrent transactions should not violate their isolation level in any execution order, thus we can enumerate all submitted orders for a group of transactions. Because the large number of transactions and their statements would lead to many submitted orders, we randomly select some of all submitted orders and test them for target DBMSs.

Algorithm 1: Deterministic transaction execution protocol under the pessimistic transaction mode.

```
   Input: subOrder
   Output: execResult
1  while !subOrder.isAllStmtsSubmitted() do
2      for i ← 1; i ≤ subOrder.length; i++ do
3          stmt ← subOrder[i]
4          if stmt.submitted then
5              continue
6          curTx ← stmt.transaction
7          if curTx.blocked then
8              continue
9          execState ← curTx.submit(stmt)
10         stmt.submitted ← True
11         if execState.blocked then
12             curTx.blocked ← True
13             execResult.execOrder.add(BlockPoint(stmt))
14             continue
15         stmt.result ← execState.getResult()
16         execResult.execOrder.add(stmt)
17         rStmts ← getResumedStmts()
18         foreach rStmt ∈ rStmts do
19             rStmt.transaction.blocked ← False
20             rStmt.result ← rStmt.execState.getResult()
21             execResult.execOrder.add(rStmt)
22         if rStmts ≠ ∅ then
23             break
24 execResult.databaseState ← getDatabaseState()
```

Transaction testing protocol. Given a submitted order for a group of transactions and their isolation level, the transaction execution process is shown in Algorithm 1. For a submitted order *subOrder*, we submit each statement in *subOrder* one by one (Line 2) to the target DBMS for execution and marks them as submitted to prevent them from being executed repeatedly (Line 9-10 and Line 4-5). For each submitted statement *stmt*, we set up an individual thread to execute *stmt*. To judge whether *stmt* is blocked, we wait for at most 2 seconds[2] for *stmt*'s execution result. If *stmt* does not return its result within 2 seconds, we determine that *stmt* is blocked, and mark *stmt* and its transaction *curTX* as blocked, and place this block point in the execution order *execOrder* (Line 11-13). We then skip the blocked statement (Line 14) and its following statements that are in the same transaction as *stmt* (Line 7-8) until *stmt* is unblocked and its transaction is resumed (Line 17-21).

If statement *stmt* is successfully executed, i.e., we can obtain its execution result within 2 seconds, we record *stmt*'s execution result and add *stmt* into *execOrder* (Line 15-16). *stmt*'s execution result can be query results for a `SELECT` statement, and reported deadlocks, warnings or errors. Note that, to thoroughly test transaction implementations, we will still submit subsequent SQL statements of a transaction even if it has reported a deadlock or error.

Once *stmt* is successfully executed, it may resume other blocked transactions. This can occur in two scenarios. First, *stmt*'s transaction has been committed or aborted, then other transactions blocked by *stmt*'s transaction can be resumed. Second, *stmt* can cause a deadlock in another transaction *tx*, and *tx* reports a deadlock and returns. Therefore, we obtain these resumed transactions, mark them as unblocked (Line 17-19), and fetch their corresponding blocked statements' execution results (Line 20-21). Then, we scan *subOrder* from the beginning and submit the un-submitted statements of transactions to the DBMS (Line 22-23 and Line 1).

Following the above test protocol, we can obtain a deterministic execution process for a group of transactions. After all transactions complete, we retrieve the database state (Line 26), and store it in the execution result *execResult*. The final transaction execution result contains all statements' execution order *execResult.execOrder*, execution status (i.e., successful, warned, blocked or failed), query results of *SELECT* statements, and the final state of the database.

4.3 Comparing Transaction Execution Results

We first execute a group of transactions for each DBMS under a certain submitted order following the protocol in Section 4.2, and then compare their execution results to find discrepancies.

Basically, we first compare the execution results for each statement in *execResult.execOrder* from beginning to end. Given two execution results of a group of transactions on two DBMSs, we fetch their *i*-th elements *stmt*1 and *stmt*2 in their execution order *execResult.execOrder*, respectively. We then compare these two statements' execution results. Note that, once we find a discrepancy, we will not continue to compare the following statements.

Inconsistent blocking. For *stmt*1 and *stmt*2, if one of them is a blocking point, while the other is not, we report a discrepancy with **inconsistent blocked statements**.

Inconsistent errors. If *stmt*1 or *stmt*2 reports a deadlock, while the other statement does not, we report a discrepancy with **inconsistent deadlock**. If one of them reports a warning or an error, but they do not report a warning or an error consistently, we report a discrepancy with **inconsistent error**. Note that, we only compare whether a warning or an error is reported, and do not compare error message, since different DBMSs can throw different error message.

Inconsistent query results. For query statements, i.e., `SELECT`, `SELECT FOR SHARE` and `SELECT FOR UPDATE`, we further compare their query results and report **inconsistent query results** if they are different. Note that DBMSs may return the data in query results in different order. We ignore the order when we compare them.

Inconsistent database final states. Regardless of whether the statement comparison results are consistent, we compare the database final state among DBMSs, i.e., whether the tables among DBMSs are consistent. If not, we report a discrepancy with **inconsistent database final state**.

5 EVALUATION

We evaluate DT^2 on three widely-used MySQL-compatible DBMSs i.e., MySQL, MariaDB and TiDB. We first explain DT^2's detection

[2]This threshold can be adjusted. However, 2 seconds are enough to judge whether *stmt* is blocked.

result in this section, and then perform an empirical study on the detected compatibility issues in Section 6.

5.1 Experimental Methodology

Tested DBMSs. We select MySQL, and two MySQL-compatible DBMSs, i.e., MariaDB and TiDB as our research subjects. Table 1 lists the details about them. The three DBMSs support the pessimistic transaction mode, and some common isolation levels. All target DBMSs are tested on the latest release versions when we started this experiment, i.e., MySQL 8.0.27, MariaDB 10.7.1, and TiDB 5.4.0.

Testing setup. Our experiment is performed on Ubuntu-20.04 with 8 CPU cores and 32 GB RAM. We build a Docker container for MySQL and MariaDB, respectively, and create a instance in it. We deploy TiDB with 2 TiDB instances, 3 TiKV instances and 3 PD instances.

Testing methodology. We run DT^2 on the three DBMSs for about one week. If a transaction test triggers a discrepancy among any two DBMSs at an isolation level, we will rerun the test under all supported isolation levels, and validate whether the discrepancy can occur under other isolation levels. Finally, we obtain 146 discrepancies in our experiment.

For each detected discrepancy, we first manually simplify its test case, e.g., removing unnecessary SQL statements, columns and data in the database, and simplifying `WHERE` expressions. Finally, we generate a simplified test case that can lead to the same discrepancy. After this process, we further remove the duplicate test cases that trigger the same discrepancies under the same isolation levels.

For each remaining discrepancy, we further investigate whether it is a bug or a compatibility issue. We investigate these discrepancies by utilizing the following process. First, we investigate the transaction implementations and user manuals provided by related DBMSs [10, 15, 23]. If a discrepancy caused by DBMSs' design choices, we classify it as a compatibility issue. Second, once we find that a discrepancy violates related DBMSs' design choices, we report it to developers as a potential bug. Note that, if we cannot fully confirm that a discrepancy is a compatibility issue, we also report it to developers for suggestions. If developers confirm that the discrepancy is a compatibility issue, we will re-classify it as a compatibility issue, otherwise, we re-classify it as a bug.

Finally, we obtain 28 bugs, and 92 compatibility issues. We discuss them in details in Section 5.2 and Section 6, respectively.

5.2 Bug Results

In total, DT^2 has detected 28 bugs as shown in Table 2. Since a bug may be triggered at more than one isolation levels, we consider the bugs with the same test cases and bug manifestation at different isolation levels as the same bug. In total, we detect 16 unique bugs.

Among the 16 unique bugs, 10 bugs are transaction bugs that can only be triggered by concurrent transactions. The remaining 6 bugs are transaction-unrelated, which can be triggered by a single SQL statement. DT^2 can detect both types of bugs, but the latter is not our focus.

Among the 10 transaction bugs, 5 bugs are revealed at `Read Uncommitted`, 7 bugs are revealed at `Read Committed`, 6 bugs are found at `Repeatable Read`, and 4 bugs are revealed at `Serializable`. Note that, a bug may be revealed at multiple isolation levels.

```
1. /*init*/ CREATE TABLE t(c1 INT PRIMARY KEY);
2. /*init*/ INSERT INTO t(c1) VALUES (3);
3. /*t1*/ BEGIN;
4. /*t1*/ UPDATE t SET c1 = 2;
5. /*t2*/ BEGIN;
6. /*t2*/ DELETE FROM t; -- t2 blocked
7. /*t1*/ UPDATE t SET c1 = 1;
8. /*t1*/ COMMIT; -- t2 released
9. /*t2*/ SELECT * FROM t FOR UPDATE;
10. /*t2*/ COMMIT;
```

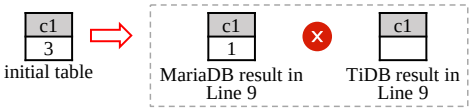

Figure 3: MariaDB#27992 reported at `Read Uncommitted`, `Read Committed`, `Repeatable Read` and `Serializable`.

We report these 16 bugs to DBMS developers, among which 6 bugs have been verified as previously-unknown bugs, one bug has been fixed (classified as verified bug), 6 bugs are duplicate, 2 bugs are false positive, and the remaining 2 bugs have not been confirmed yet. 4 newly found bugs are of high priority. In MariaDB, 2 transaction bugs are marked as critical and 2 transaction-unrelated bugs are marked as major. In TiDB, 2 bugs are marked as moderate.

For the 6 newly detected bugs, 2 bugs leave the database in an incorrect state, one bug causes the query result to be incorrect, 2 bugs lead to miss expected errors, and the remaining bug reports an unexpected error.

5.3 Interesting Bugs

MariaDB#MDEV-27992 [7]. Figure 3 shows the simplified test case that triggers a new transaction bug in MariaDB at all isolation levels. The initial table is shown in the table on the left, where $c1$ is the primary key. $t1$ first updates value in $c1$ to 2 (Line 4), while $t2$ simultaneously deletes all rows in the table and is blocked (Line 6). $t1$ then updates value in $c1$ to 1 and is committed (Line 7-8). After that, $t2$ is resumed and queries the database state (Line 9). The database states of MariaDB and TiDB are different, as shown in the two tables on the right. MariaDB returns [(1)], while TiDB returns the empty table due to the `DELETE` operation in $t2$ Line 6. DT^2 detects this inconsistent query results. The states of the two databases after $t2$ is committed are the same as the results queried in Line 9. Therefore this discrepancy causes **inconsistent database final states**. The root cause of this bug is that MariaDB does not `UPDATE` primary keys atomically. This bug will be fixed in the upcoming release MariaDB 10.8.3.

```
1. /*init*/ CREATE TABLE t(c1 INT PRIMARY KEY, c2
     INT);
2. /*init*/ INSERT INTO t(c1, c2) VALUES (1, 1);
3. /*t1*/ BEGIN;
4. /*t1*/ UPDATE t SET c1 = 2, c2 = 2;
5. /*t2*/ BEGIN;
6. /*t2*/ DELETE FROM t; -- t2 blocked
7. /*t1*/ COMMIT;
8. /*t2*/ SELECT * FROM t; -- [(1, 1)] in TiDB
9. /*t2*/ COMMIT;
```

Listing 1: TiDB#33315 reported at `Repeatable Read`.

Table 2: Bugs detected by DT^2.

DBMS	Transaction Related				Transaction Unrelated	Total (Unique*)	Confirmed		Unconfirmed	False Positive
	RU	RC	RR	SER			Verified	Duplicate		
MySQL	2	2	0	1	1	3(4)	0	2	0	2
MariaDB	3	3	3	3	3	15(7)	4	1	2	0
TiDB	-	2	3	-	2	7(5)	2	3	0	0
Total	5	7	6	4	6	28(16)	6	6	2	2

* The numbers in parentheses show unique bugs.

Table 3: Compatibility issues detected by DT^2.

DBMS	Detected						Unique					
	Transaction Related				Transaction Unrelated	Total	Transaction Related				Transaction Unrelated	Total
	RU	RC	RR	SER			RU	RC	RR	SER		
MySQL-MariaDB	0	0	5	5	2	12	0	0	4	4	2	10
MySQL-TiDB	-	16	36	-	0	52	-	11	28	-	0	39
MariaDB-TiDB	-	16	36	-	2	54	-	11	30	-	2	43
Total	0	32	77	5	4	118	0	22	62	4	4	92

TiDB#33315 [26]. Listing 1 illustrates a transaction bug detected in TiDB at `Repeatable Read`. In this test case, MySQL and MariaDB have the same execution results, while TiDB's query result is different from theirs. During database generation, $c1$ is set as the primary key. $t1$ first updates values in $c1$ and $c2$ to 2 (Line 4). Then $t2$ tries to delete the table and is blocked (Line 6). After $t1$ is committed (Line 7), $t2$ executes `DELETE` successfully and retrieves the table by `SELECT` statement (Line 8). MySQL and MariaDB return an empty set, whereas TiDB returns [(1, 1)] wrongly. After two transactions are committed, the database states are consistent in the three DBMSs. DT^2 reports the discrepancy with **inconsistent query results**. This bug occurs only when using clustered index on $c1$. When $t2$ performs `DELETE` in Line 6, instead of marking (1, 1) and (2, 2) as deleted, it only marks updated values (2, 2) as deleted. Thus, the `SELECT` statement in Line 8 reads snapshot and returns [(1, 1)]. TiDB developers mark this bug as duplicate, but the scenario that triggers the bug are different.

```
1. /*init*/ CREATE TABLE t(c1 INT PRIMARY KEY, c2
     INT);
2. /*init*/ INSERT INTO t(c1, c2) VALUES (1, 1), (2,
     2);
3. /*t1*/ BEGIN;
4. /*t1*/ SELECT * FROM t; -- [(1, 1), (2, 2)]
5. /*t2*/ BEGIN;
6. /*t2*/ DELETE FROM t WHERE c1 = 1; -- t2 blocked
7. /*t1*/ DELETE FROM t; -- t2 encounters deadlock
     in MySQL
8. /*t1*/ COMMIT;
9. /*t2*/ COMMIT;
```

Listing 2: MySQL#106655 reported at `Serializable`.

MySQL#106655 [24]. Listing 2 shows a transaction bug detected in MySQL by differentially testing MySQL and MariaDB at `Serializable`. When `SELECT` statement is executed, shared locks are taken to prevent other transactions from modifying the data that has been read. Thus, after $t1$ reads the initial table (Line 4), $t2$ is blocked since it tries to delete the rows where $c1$ is 1 (Line 6). While $t1$ executes the next statement that deletes the table (Line 7), a deadlock occurs in MySQL but does not occur in MariaDB. Therefore, DT^2 reports the test report with **inconsistent deadlock**. This bug is caused by a failed escalation of a lock from the shared one to the exclusive one in MySQL. This is confirmed as a duplicate bug. This bug is fixed in the upcoming release 8.0.29.

```
1. /*init*/CREATE TABLE t(c1 BLOB NOT NULL,c2 TEXT);
2. /*init*/INSERT IGNORE INTO t VALUES(NULL, NULL),
     (NULL, 'aa');
3. UPDATE t SET c2 = 'test' WHERE c1;
   -- MariaDB reports an error
   ERROR: Truncated incorrect DOUBLE value: ''
```

Listing 3: MariaDB#28140 reported as a transaction-unrelated bug.

MariaDB#MDEV-28140 [25]. Although our purpose is to test transactions, DT^2 can also detect bugs that are not related to transactions, i.e., bugs are caused by one statement in transactions. Listing 3 shows a new bug detected in MariaDB. In this test case, MariaDB reports an error when executing only the `UPDATE` statement (Line 3) whereas MySQL does not report an error, thus DT^2 reports **inconsistent error**. MariaDB developers confirm it as a bug.

5.4 False Positives

Two of our reported bugs to MySQL are considered as not a bug. However, we still consider that these test cases can indicate some issues. We simply explain them as follows.

MySQL#106629 [8]. This problem is triggered by the same test case in Figure 3, and its manifestation is the same as that in MariaDB. However, MySQL developers think that the behavior is expected.

```
1. /*init*/ CREATE TABLE t(c1 VARCHAR(10));
2. /*init*/ INSERT IGNORE INTO t(c1) VALUES ('try');
3. UPDATE t SET c1 = 'test' WHERE (CAST(('a12') AS
     DOUBLE)) IS NULL;
4. UPDATE t SET c1 = 'test' WHERE (CAST(('a12') AS
     DOUBLE)) IS NOT NULL; -- MySQL reports an error
   ERROR: Truncated incorrect DOUBLE value: 'a12'
```

Listing 4: MySQL#107125 reported as a false positive.

MySQL#107125 [27]. Listing 4 illustrates a problem detected in MySQL. In this test case, UPDATE statement with IS NULL does not return error or warning (Line 3), since the WHERE expression with IS NULL is evaluated to be FALSE. However, UPDATE statement with IS NOT NULL returns an error (Line 4), even if the WHERE expression with IS NOT NULL is evaluated to be TRUE. MySQL developers think it is not a bug. They explain that *"IS NULL and IS NOT NULL have different, non-symmetric validation functions"*. But they also said *"It could be reasonable to treat IS NOT NULL the same as IS NULL, though"*.

6 COMPATIBILITY ISSUE STUDY

As shown in Table 3, we detect 118 test scenarios that can trigger compatibility issues in our experiment (Detected). After removing duplicate scenarios in Section 5.1, we finally obtain 92 unique compatibility issues (Unique), in which 88 are transaction-related compatibility issues. In the following, we mainly study these 88 transaction-related compatibility issues.

For the 88 transaction-related compatibility issues, 8 issues are found between MySQL and MariaDB, 39 issues are found between MySQL and TiDB, and 41 issues are found between MariaDB and TiDB. This distribution illustrates that TiDB has more compatibility issues than MariaDB.

Among the 88 compatibility issues, none are found at Read Uncommitted, 22 issues are found at Read Committed, 62 issues are found at Repeatable Read, and 4 issues are found at Serializable.

We further investigate these transaction-related compatibility issues, and try to answer two research questions.

- **RQ1 (Root cause):** What are the root causes of compatibility issues?
- **RQ2 (Consequence):** What consequences do compatibility issues have?

Study methodology. To reduce the subjective bias, three authors independently investigate these compatibility issues, and identify their root causes and consequences. Then, they discuss analysis results, and reach consensus for each compatibility issue.

6.1 RQ1. Root Cause

From the 88 transaction-related compatibility issues, we identify three root causes. We elaborate them as follows.

Inconsistent lock mechanisms in DBMSs. Different DBMSs adopt different lock mechanisms for different SQL statements, e.g., INSERT, UPDATE, and DELETE, at different isolation levels. We briefly explain some key differences as follows.

- MySQL. At *Read Committed*, MySQL performs *semi − consistent* read, in which, when an UPDATE statement examines a row that is already locked, MySQL first utilizes the latest committed version to determine whether the row matches the WHERE condition. If yes, MySQL will lock the row. Otherwise, MySQL will not lock the row. However, at Repeatable Read and Serializable isolation levels, MySQL will lock its examined rows regardless of the WHERE condition's evaluated value. MySQL further utilizes Gap Lock at Repeatable Read and Serializable to lock a gap between index records.
- MariaDB. MariaDB's lock mechanism is almost the same as MySQL. They have slightly different lock behaviors for some cases, e.g., locks for primary keys.
- TiDB. At both Read Committed and Repeatable Read, TiDB's lock mechanism for all kinds of SQL statements (e.g., UPDATE and DELETE) is similar to *semi − consistent* read in MySQL. TiDB does not support Gap Lock.

```
1./*init*/ CREATE TABLE t(c1 INT, c2 INT);
2./*init*/ INSERT INTO t(c1, c2) VALUES (2, NULL);
3./*t1*/ BEGIN;
4./*t1*/ UPDATE t SET c1 = 1, c2 = 1;
5./*t2*/ BEGIN;
6./*t2*/ UPDATE t SET c1 = 3 WHERE c2; -- t2
     blocked in MySQL, not in TiDB
7./*t1*/ COMMIT;
8./*t2*/ COMMIT;
```

Listing 5: Inconsistent lock point at Repeatable Read.

The lock mechanism differences can lead to different execution behaviors, which can cause compatibility issues. 78 (89%) compatibility issues are caused by inconsistent lock mechanisms. Listing 5 shows a compatibility issue caused by inconsistent lock mechanisms at Repeatable Read for MySQL and TiDB. The initial table contains only one row [(2, null)]. $t1$ firstly updates the values in $c1$ and $c2$ to 1 (Line 4), and $t2$ tries to update the row where $c2$ is not NULL (Line 6). As a result, $t2$ is blocked in MySQL. After $t1$ is committed (Line 7), $t2$ is resumed and updates the row successfully. In TiDB, $t2$ first evaluates the WHERE condition (i.e., WHERE c2) on the row [(2, null)] to be FALSE, so it is not blocked and executes UPDATE without affecting any data. The database final states between MySQL and TiDB are different. In MySQL, the database state is [(3, 1)], while it is [(1, 1)] in TiDB, which causes inconsistent database states.

The aborted transaction is different when a deadlock occurs. 4 (5%) compatibility issues are due to inconsistent aborted transaction after a deadlock occurs. After a deadlock occurs, one of transactions is rolled back to break the deadlock. However, DBMSs adopt different strategies to choose the aborted transaction. Therefore, different DBMSs can abort different transactions when a deadlock occurs.

```
1. /*init*/ CREATE TABLE t(c1 INT, c2 INT);
2. /*init*/ INSERT INTO t(c1, c2) VALUES (1, 1);
3. /*t1*/ BEGIN;
4. /*t2*/ BEGIN;
5. /*t2*/ DELETE FROM t WHERE c1 = 1;
6. /*t2*/ COMMIT;
7. /*t1*/ SELECT * FROM t; -- [] in MySQL and
     MariaDB, [(1, 1)] in TiDB
8. /*t1*/ COMMIT;
```

Listing 6: Inconsistent snapshot creation.

Inconsistent snapshot creation. 6 (7%) compatibility issues are caused by inconsistent snapshot creation location at Repeatable Read. In TiDB, snapshot is established at the BEGIN statement by default. However, in MySQL and MariaDB, snapshot is created by the first SELECT by default.

Listing 6 illustrates a compatibility issue caused by inconsistent snapshot creation at Repeatable Read. The initial table contains one row [(1, 1)]. $t1$ first starts (Line 3). Then, $t2$ starts, deletes the row where $c1$ is 1 and is committed (Line 4-6). Finally, $t1$ retrieves the

Table 4: Root causes and consequences of transaction-related compatibility issues.

DBMS	Root Cause			Consequence			Total
	Lock mechanism	Aborted transaction	Snapshot creation	Database state	Query result	No difference	
MySQL-MariaDB	8	0	0	4	0	4	8
MySQL-TiDB	34	2	3	16	9	14	39
MariaDB-TiDB	36	2	3	17	9	15	41
Total	78	4	6	37	18	33	88

database with SELECT statement (Line 7). TiDB reads the snapshot established in Line 3, and returns [(1, 1)]. Whereas, MySQL and MariaDB read the snapshot created by the first SELECT (i.e., Line 7), and return []. After $t2$ is committed (Line 8), the database states are consistent, i.e., [].

6.2 RQ2. Consequence

As shown in Table 4, for the 88 transaction-related compatibility issues, 66 (75%) can cause inconveniences among DBMSs, i.e., inconsistent database states and query results. The remaining 22 (25%) compatibility issues can cause inconsistent blocked statements, but do not affect the final execution results.

Inconsistent database state. In 37 (42%) compatibility issues, the same transactions can result in inconsistent database states among the tested DBMSs. Listing 5 shows such a compatibility issue.

Inconsistent query result. For the remaining 51 compatibility issues that cannot result in inconsistent database states, 18 (21%) compatibility issues result in inconsistent query results for SELECT statements. Listing 6 shows such a compatibility issue. Note that, if a compatibility issue can cause inconsistent database state, we will not count it as an issue with inconsistent query result.

6.3 Lessons Learned

Insufficiency over transaction behaviors. Although MariaDB and TiDB claim to be compatible with MySQL, their different behaviors of transaction executions are not well documented, which introduces many compatibility issues at different isolation levels. Although isolation levels have been clearly specified in literatures [1, 28, 29, 34], transaction execution behaviors can be seriously affected by various design choices in different DBMSs, e.g., lock mechanisms and snapshot creation. From our study, we can see that there lacks of a specification for transaction behaviors. This may cause confusion for DBMS developers and DBMS application developers. We hope a transaction behavior specification can significantly alleviate this situation.

Guidance on database migration. Transaction-related compatibility issues can cause inconsistent query results and inconsistent database states. DBMS application developers should be aware of these compatibility issues when migrating their applications among DBMSs. They need to consider whether these compatibility issues can break down their applications slightly. The compatibility issues that we revealed can be used to analyze the effect on their applications when migrating their applications among these DBMSs.

7 DISCUSSION

In this section, we discuss the threats and limitations in our work.

7.1 Threats to Validity

First, we evaluate DT^2 using three MySQL-compatible DBMSs. Our studied MySQL-compatible DBMSs are widely-used and provide mainstream isolation levels. We believe they are representative for MySQL-compatible DBMSs. However, our experimental results may not reflect the situation in other DBMSs, e.g., PostgreSQL-compatible DBMSs and their optimistic transactions.

Second, we may introduce human errors when manually analyzing and determining whether a discrepancy is a compatibility issue. To alleviate this threat, three authors carefully study each discrepancy found during testing. If we cannot reach consensus for a discrepancy, we ask DBMS developers for confirmation.

Third, DT^2 adopts random testing approach, and may not reveal all compatibility issues among the tested DBMSs. Thus, our study on compatibility issues may not be complete.

7.2 Limitations

Support for more DBMSs. Currently DT^2 only supports three MySQL-compatible DBMSs, but it can be extended to support other DBMSs. We also intend to support more MySQL-compatible DBMSs. DT^2 can support other MySQL-compatible DBMSs by avoiding to generate test cases involving features that these DBMSs do not support. DT^2 can also extended to support other kinds of DBMSs, e.g., PostgreSQL-compatible DBMSs. For supporting these DBMSs, we have to re-implement database generation and SQL statement generation.

Support for the optimistic transaction mode. Some DBMSs support the optimistic transaction mode, e.g., PostgreSQL and TiDB. In the optimistic mode, a transaction is blocked, then checks for conflicts and rolls back conflicting transactions before it is committed. Testing DBMSs under this mode is similar to testing DBMSs under the pessimistic transaction mode. To support optimistic mode, we need to submit transaction statements to DBMS in a submitted order, record aborted transactions and add them to the test results.

8 RELATED WORK

Differential testing. Differential testing [55] has been widely applied in many domains, such as compilers [32, 65], runtime systems [44, 45], symbolic execution engines [51], and software libraries [46]. For DBMSs, RAGS [61] adopts differential testing of single query statements to find database bugs. Jung et al. [50] introduces a tool, APOLLO, to detect performance regressions by executing

SQL queries on two versions of one DBMS. Sotiropoulos et al. [63] applies differential testing on Object-Relational Mapping systems to find ORM-specific bugs. Zheng et al. [67] proposes a differential testing approach, Grand, for detecting logic bugs in Gremlin-based graph database systems. Inspired by these works, we utilize differential testing to detect transaction bugs and transaction-related compatibility issues.

Database testing. Many approaches that automatically generate SQL queries and databases to test DBMSs have been proposed. SQLsmith [18] generates random SQL queries to find DBMS bugs. More recently, Rigger et al. [58–60] has proposed a series of works to find logical bugs by generating single SQL queries. Bati et al. [33] presents the technique of randomly generating test cases and using the execution feedback obtained from the tested DBMS to generate queries. These existing approaches are designed to find DBMS-specific bugs and cannot detect transactions bugs.

Transaction testing and verification. To improve the correctness of transaction implementations, many works verify whether DBMSs violate their claimed transactional consistency and isolation. Brutschy et al. [41] proposes an effective serializability criterion to extend the serializability of conflicts to eventually consistency semantics. They also present a dynamic analyzer to check whether a given program execution conforms to the criterion. Elle [53] and COBRA [64] verify the serializability of database based on dependency graph introduced by Adya [29]. However, these works mainly focus on designing specific $key - value$ database models, and can not be applied to test complex transaction features in real-world DBMSs, e.g., database constraints and cross-table queries.

Isolation violations in database-backed applications. Some works have been proposed to detect or debug isolation violations and anomalies in real-world database-backed applications. Deng et al. [48] designs AGENDA for testing database-driven applications. Rahmani et al. [56] introduces a static testing framework, CLOTHO, to detect serializability violations in database-backed applications running on weakly-consistent storage systems. Gan et al. [49] presents a tool, IsoDiff, for debugging anomalies that are caused by `Read Committed` and `Snapshot Isolation` isolation levels in real-world applications. Biswas et al. [38] proposes a mock storage system, MonkeyDB, to test the correctness of storage-backed applications at weak isolation levels. Our compatibility issue study among DBMSs can be used to further detect violations in database-based applications.

9 CONCLUSION

Buggy transaction implementations in DBMSs can violate their claimed ACID properties, and lead to severe consequences, e.g., incorrect database states and query results. In this paper, we propose DT^2, an automated transaction testing approach to detect transaction discrepancies among DBMSs by differential testing. We evaluate DT^2 on three widely-used MySQL-compatible DBMSs, and have detected 10 unique transaction bugs and 88 transaction-related compatibility issues from the detected discrepancies.

ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China (62072444, 61732019), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences (2018142, 201924).

REFERENCES

[1] 2022. The ANSI isolation levels. http://www.adp-gmbh.ch/ora/misc/isolation_level.html.
[2] 2022. CockroachDB. https://www.cockroachlabs.com.
[3] 2022. Data Types in MariaDB. https://mariadb.com/kb/en/data-types.
[4] 2022. Data Types in MySQL. https://dev.mysql.com/doc/refman/8.0/en/data-types.html.
[5] 2022. Data Types in TiDB. https://docs.pingcap.com/tidb/stable/data-type-overview.
[6] 2022. DB-Engines. https://db-engines.com/en/ranking.
[7] 2022. DELETE fails to delete record after blocking is released. https://jira.mariadb.org/browse/MDEV-27992.
[8] 2022. DELETE fails to delete record after blocking is released. https://bugs.mysql.com/106629.
[9] 2022. GitHub. https://github.com.
[10] 2022. InnoDB Transaction Model. https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-model.html.
[11] 2022. INSERT fails to return an error after transaction abort. https://jira.mariadb.org/browse/MDEV-27922.
[12] 2022. MariaDB. https://mariadb.org.
[13] 2022. MySQL. https://www.mysql.com.
[14] 2022. PostgreSQL. https://www.postgresql.org.
[15] 2022. Set Transaction in MariaDB. https://mariadb.com/kb/en/set-transaction.
[16] 2022. SQLancer. https://www.manuelrigger.at/dbms-bugs.
[17] 2022. SQLite. https://www.sqlite.org/index.html.
[18] 2022. SQLsmith. https://jepsen.io.
[19] 2022. TiDB Optimistic Transaction Model. https://docs.pingcap.com/tidb/stable/optimistic-transaction.
[20] 2022. TiDB Pessimistic Transaction Mode. https://docs.pingcap.com/tidb/stable/pessimistic-transaction.
[21] 2022. TiDB, PingCAP. https://pingcap.com.
[22] 2022. Transaction Isolation Levels in MySQL. https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html.
[23] 2022. Transactions in TiDB. https://docs.pingcap.com/tidb/stable/transaction-overview.
[24] 2022. Unexpected Deadlock Happened When Two transaction Execute Concurrently. https://bugs.mysql.com/106655.
[25] 2022. Unexpected error when UPDATE a NULL. https://jira.mariadb.org/browse/MDEV-28140.
[26] 2022. Weird SELECT when table has the primary key. https://github.com/pingcap/tidb/issues/33315.
[27] 2022. Weird statement with IS NULL and with IS NOT NULL. https://bugs.mysql.com/107125.
[28] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D. Dissertation. Massachusetts Institute of Technology.
[29] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of International Conference on Data Engineering (ICDE)*. 67–78.
[30] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *Proceedings of the VLDB Endowment* 7, 3 (2013), 181–192.
[31] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Transactions on Database Systems* 41, 3 (2016).
[32] Gerg"o Barany. 2018. Finding Missed Compiler Optimizations by Differential Testing. In *Proceedings of International Conference on Compiler Construction*. 82–92.
[33] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. 2007. A Genetic Approach for Random Testing of Database Systems. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 1243–1251.
[34] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Vol. 24. 1–10.
[35] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.
[36] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. 2007. QAGen: Generating Query-Aware Test Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 341–352.

[37] Ranadeep Biswas and Constantin Enea. 2019. On the Complexity of Checking Transactional Consistency. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 165:1–165:28.

[38] Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. 2021. MonkeyDB: Effectively Testing Correctness under Weak Isolation Levels. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 132:1–132:27.

[39] Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible Database Generators. In *Proceedings of the VLDB Endowment*. 1097–1107.

[40] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. 2006. Generating Queries with Cardinality Constraints for DBMS Testing. In *IEEE Transactions on Knowledge and Data Engineering*, Vol. 18. 1721–1725.

[41] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2017. Serializability for Eventual Consistency: Criterion, Analysis, and Applications. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Vol. 52. 458–472.

[42] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *Proceedings of International Conference on Concurrency Theory (CONCUR)*. 58–71.

[43] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A Structured English Query Language. In *Proceedings of ACM SIGFIDET Workshop on Data Description, Access and Control*. 249–264.

[44] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep Differential Testing of JVM Implementations. In *Proceedings of International Conference on Software Engineering (ICSE)*. 1257–1268.

[45] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-Directed Differential Testing of JVM Implementations. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 85–99.

[46] Yuting Chen and Zhendong Su. 2015. Guided Differential Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 793–804.

[47] Edgar F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387.

[48] Yuetang Deng, Phyllis Frankl, and David Chays. 2005. Testing Database Transactions with AGENDA. In *Proceedings of International Conference on Software Engineering (ICSE)*. 78–87.

[49] Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. 2020. IsoDiff: Debugging Anomalies Caused by Weak Isolation. *Proceedings of the VLDB Endowment (PVLDB)* 13, 12 (2020), 2773–2786.

[50] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *Proceedings of the VLDB Endowment (VLDB)* 13, 1 (2019), 57–70.

[51] Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 590–600.

[52] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O Laleye, and Sarfraz Khurshid. 2008. Query-Aware Test Generation Using a Relational Constraint Solver. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 238–247.

[53] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. In *Proceedings of the VLDB Endowment*, Vol. 14. 268–280.

[54] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[55] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[56] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. 2019. CLOTHO: Directed Test Generation for Weakly Consistent Database Systems. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 117:1–117:28.

[57] David Patrick Reed. 1978. *Naming and synchronization in a decentralized computer system*. Technical Report.

[58] Manuel Rigger and Zhendong Su. 2020. 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)*. 1140–1152.

[59] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 211:1–211:30.

[60] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 667–682.

[61] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the VLDB Endowment*. 618–622.

[62] Xiaohui Song and Jane W-S Liu. 1990. Performance of multiversion concurrency control algorithms in maintaining temporal consistency. In *Proceedings of Annual International Computer Software and Applications Conference*. 132–139.

[63] Thodoris Sotiropoulos, Stefanos Chaliasos, Vaggelis Atlidakis, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Data-Oriented Differential Testing of Object-Relational Mapping Systems. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 1535–1547.

[64] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 63–80.

[65] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 283–294.

[66] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of International Conference on Management of Data (SIGMOD)*. 1629–1642.

[67] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. 2022. 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)*. 302–313.