Simple Testing Can Expose Most Critical Transaction Bugs: Understanding and Detecting Write-Specific Serializability Violations in Database Systems

Ziyu Cui*
Wensheng Dou*†‡
Institute of Software at CAS, China
{cuiziyu20,wsdou}@otcaix.iscas.ac.cn

Yu Gao*
Rui Yang*
Institute of Software at CAS, China
{gaoyu15,yangrui22}@otcaix.iscas.ac.cn

Yingying Zheng*
Institute of Software at CAS, China
zhengyingying14@otcaix.iscas.ac.cn

Jiansen Song*
Institute of Software at CAS, China
songjiansen20@otcaix.iscas.ac.cn

Yuan Feng
Wuhan Dameng Database Co., Ltd.
fengyuan@dameng.com

Jun Wei*†
Institute of Software at CAS, China
wj@otcaix.iscas.ac.cn

ABSTRACT

Database Management Systems (DBMSs) utilize transactions to guarantee data consistency and integrity. Incorrect implementations of transaction processing mechanisms can introduce critical transaction bugs, which can lead to incorrect database states after the involved transactions complete. However, we lack an effective test oracle to determine whether a DBMS produces a correct database state for a given concurrent transaction schedule.

In this paper, we propose a general property for concurrent transaction schedules, *write-specific serializability*, in which a schedule of concurrent transactions should produce the same database state as a corresponding serial schedule of the same transactions. Through our empirical study on 35 critical transaction bugs collected from six widely-used DBMSs, we find that write-specific serializability can be an effective test oracle to expose critical transaction bugs in DBMSs. We further develop a simple and general transaction testing approach, `WriteCheck`, to automatically detect write-specific serializability violations by identifying inconsistencies in the final database states produced by the original transaction schedule and its corresponding serial schedule. We evaluate `WriteCheck` on the latest versions of six production-grade DBMSs, and have found 22 write-specific serializability violations, 11 of which have been confirmed as new critical transaction bugs.

PVLDB Reference Format:
Ziyu Cui, Wensheng Dou, Yu Gao, Rui Yang, Yingying Zheng, Jiansen Song, Yuan Feng, and Jun Wei. Simple Testing Can Expose Most Critical Transaction Bugs: Understanding and Detecting Write-Specific Serializability Violations in Database Systems. PVLDB, 18(8): 2547 - 2560, 2025.
doi:10.14778/3742728.3742747

*Affiliated with Key Lab of System Software at CAS, State Key Lab of Computer Science at Institute of Software at CAS, and University of CAS, Beijing. CAS is the abbreviation of Chinese Academy of Sciences.
†Affiliated with Nanjing Institute of Software Technology, University of CAS, Nanjing.
‡Wensheng Dou is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:10.14778/3742728.3742747

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tcse-iscas/WriteCheck.

1 INTRODUCTION

Database Management Systems (DBMSs) have been widely used to efficiently store and retrieve data in many applications. Many DBMSs, e.g., MySQL [10], PostgreSQL [12], SQLite [15], MariaDB [9], CockroachDB [2] and TiDB [18], utilize the relational data model [35] to organize data, and Structured Query Language (SQL) [33] to access relational data. Specifically, DBMSs leverage read operations (e.g., `SELECT`) to retrieve data, and write operations (e.g., `INSERT`, `UPDATE` and `DELETE`) to modify data.

DBMSs utilize transaction processing mechanisms to guarantee data consistency and integrity. A transaction is an indivisible unit of work that should be executed as a whole. An *explicit* transaction consists of a transaction starting statement (e.g., `BEGIN`), some SQL statements that retrieve and modify data, and a transaction ending statement (e.g., `COMMIT` and `ROLLBACK`). When the *autocommit* mode is enabled, each SQL statement that is not wrapped in explicit transactions, i.e., autocommit statement, forms an implicit transaction on its own. We name an autocommit statement as an ACS transaction for easy presentation.

DBMSs ensure that transactions satisfy ACID properties [28, 36, 39, 49, 66] despite errors and mishaps [1, 19, 20, 23–25, 32]. However, incorrect implementations of transaction processing mechanisms in DBMSs can introduce transaction bugs (`txBugs` for short) that violate the claimed ACID properties. In this paper, we mainly focus on `txBugs` that lead to incorrect database states after the involved transactions complete. To distinguish these `txBugs` from others, we call them as *critical* `txBugs`[1]. Critical `txBugs` in DBMSs are difficult to detect automatically. A key challenge is to come up with an effective test oracle, which can determine whether a DBMS produces a correct database state for a concurrent transaction schedule.

Existing transaction verification [28, 34, 49, 66] and testing [36, 39, 45] approaches can be used to detect critical `txBugs`. However, these approaches have obvious limitations. Transaction verification

[1] `txBugs` can also lead to other consequences, e.g., wrong query results and performance degradation. Once the involved transactions complete, the database states remain correct. These `txBugs` can also be highly valued by DBMS developers.

approaches (e.g., Biswas et al. [28], Emme [34], Elle [49] and Cobra [66]) can only work on simple *key-value*-like data models, and can only detect isolation bugs (a subset of txBugs) that violate an isolation level claimed by the target DBMS. They cannot support non-*key-value*-like data models and complex SQL features (e.g., sub queries and complex predicates), which are commonly used in DBMSs. Transaction testing approaches [36, 39, 45] leverage differential testing and *sophisticated* transaction oracle construction to detect txBugs. However, these approaches can only work on common features supported by multiple DBMSs [36], or support limited SQL features [39, 45], or are impaired by incorrect statement dependency inference [45] (more details in Section 6.5). Therefore, existing approaches can miss many critical txBugs.

Write-specific serializability. DBMSs usually support multiple isolation levels, e.g., `Read Committed`, `Repeatable Read` and `Serializable` in MySQL. We observe that, for these isolation levels, a transaction cannot overwrite a data item that has previously been written by another in-flight transaction [1, 19, 20, 25]. Therefore, concurrent transactions that contain write-write conflicting operations should be blocked or aborted. This observation motivates us to propose *write-specific serializability* (WSS for short), a general property for write operations in concurrent transactions: *A schedule $\mathcal{H}_{con}$ of concurrent transactions should produce the same final database state after executing $\mathcal{H}_{con}$ as that produced by a corresponding serial schedule $\mathcal{H}_{ser}$ of the same transactions.* Otherwise, a WSS violation occurs, and $\mathcal{H}_{con}$ will produce an incorrect final database state, i.e., causing a critical txBug.

WSS is different from conflict serializability [70], in which a schedule of concurrent transactions should produce *the same effect for all operations (e.g., read and write)* and the same database state as a corresponding serial schedule of the same transactions. WSS only focuses on *the effect of write operations*, and ignores read operations. Therefore, WSS is a weaker property than conflict serializability. However, WSS can be applied on all isolation levels defined by Adya [19], while conflict serializability can only be applied on the `Serializable` isolation level.

Empirical study on WSS violations. To better understand WSS violations and WSS's capability to expose critical txBugs, we conduct the first empirical study on 35 real-world critical txBugs, which are collected from six widely-used DBMSs, i.e., MySQL, PostgreSQL, SQLite, MariaDB, CockroachDB and TiDB. We thoroughly analyze these critical txBugs, and obtain several interesting findings.

- In most (91.4%) critical txBugs, their transaction test cases violate WSS. This indicates that WSS can be an effective test oracle to expose critical txBugs in DBMSs.
- All studied WSS violations can be triggered by small transaction test cases, and follow the small scope hypothesis [43]. All WSS violations require no more than two initial tables, and no more than five transactions. Almost all (96.0%) involved transactions contain no more than five statements.
- Most (90.6%) WSS violations can be triggered by deterministically executing the SQL statements in the transaction test cases in a certain order.

WSS violation detection. Based on these findings, we further develop `WriteCheck`, a simple and general transaction testing approach to detect WSS violations. We first generate a deterministic transaction test case, which contains an initial database, a group of transactions, and a submitted order for all the statements in the generated transactions. We then submit the statements in the transactions to the target DBMS by following the submitted order, and obtain the concurrent transaction schedule and the final database state. We further infer the corresponding serial schedule under WSS for the involved transactions by analyzing the concurrent schedule, and then obtain the database state after applying the serial schedule on the same initial database. We compare the final database states produced by the original concurrent schedule and the serial schedule. If we find any inconsistency, we detect a WSS violation.

We implement `WriteCheck` based on SQLancer [14], and extend SQLancer to detect WSS violations. `WriteCheck` can work for different isolation levels [1, 19, 20, 25] and concurrency control modes [26, 50, 57, 65, 73]. To demonstrate the effectiveness of `WriteCheck`, we have evaluated it on the latest versions of the six DBMSs in our empirical study. In total, we have detected 22 unique WSS violations, 11 of which have been confirmed as new critical txBugs. For the 29 critical txBugs reported by `WriteCheck` and the state-of-the-art transaction testing approaches [36, 39, 45], `WriteCheck` can expose all these txBugs, while existing approaches [36, 39, 45] can expose at most 23 txBugs. This shows that `WriteCheck` can expose and detect more critical txBugs than the state-of-the-art approaches.

Contributions. We make the following contributions.

- We are the first to propose write-specific serializability for concurrent transactions, which can expose most critical transaction bugs that can cause concurrent transactions to produce incorrect final database states.
- We present the first empirical study on real-world write-specific serializability violations, and obtain some interesting findings.
- We propose a simple and general transaction testing approach `WriteCheck` to detect write-specific serializability violations.
- We implement `WriteCheck` and apply it on six popular DBMSs, and have detected 11 new critical transaction bugs.

2 WRITE-SPECIFIC SERIALIZABILITY

2.1 Illustrative Example

Figure 1a shows a transaction test case that triggers a real-world WSS violation TiDB#42121, which we detected in TiDB at the `Repeatable Read` isolation level. In this test case, transaction T_1 and T_2 are concurrently executed by following the order of the arrows. TiDB's implementation does not block the `DELETE` statement s_{13} that conflicts with the `REPLACE` statement s_{23}[2], and only deletes the row with $c1 = 1.0$. Therefore, the schedule in Figure 1a leads to an incorrect database state after T_1 and T_2 complete, i.e., {2.0} in table t. For the correct implementation, s_{13} should be blocked until T_2 is committed, and then deletes all rows in which $c1 < 5.0$. TiDB developers have classified this violation as *critical* and fixed it quickly.

The serial schedule of Figure 1a is $T_2 \rightarrow T_1$ (Figure 1b), and produces a correct database state, i.e., an empty set for table t, which is different from the database state in the original schedule in Figure 1a. Thus, a WSS violation occurs.

[2]Since the `REPLACE` statement s_{23} updates the row with $c1 = 1.0$ but does not change its value, TiDB's implementation forgets to mark the row with $c1 = 1.0$ to be locked in s_{23}. Note that s_{13} does not conflict with s_{22}, since TiDB does not support gap lock [6].

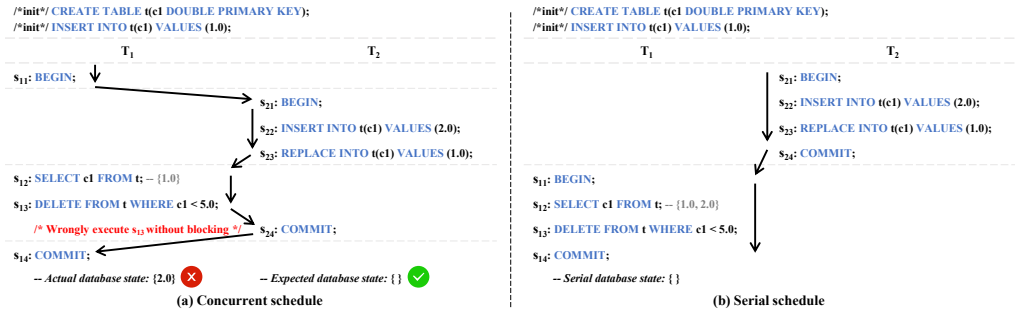

Figure 1: A transaction test case triggers a write-specific serializability violation TiDB#42121 in TiDB. This violation occurs at the `Repeatable Read` isolation level under the pessimistic transaction mode, and leads to an incorrect database state.

2.2 WSS Definition

Let $T_i = [s_{i_1}, s_{i_2}, ..., s_{i_m}]$ (in which m is the number for the SQL statements in T_i) be an explicit transaction (starting with a `BEGIN` statement, followed by some data query and manipulation statements (i.e., DQL and DML statements), and ending with a `COMMIT` or `ROLLBACK` statement) or an ACS transaction (containing only one autocommit DQL / DML statement, and $m = 1$). We restrict that each write operation (i.e., DML statement) within transaction T_i must be independent and should not rely on the values read in earlier read operations (i.e., DQL statements)[3]. A write operation can either write explicit values (e.g., INSERT t(c1) VALUES (10)) or utilize the values read by itself (e.g., UPDATE t SET c1 = c1 + 1).

Let $\mathcal{T} = \{T_1, T_2, ..., T_n\}$ be the concurrent transactions, and n be the number of the concurrent transactions. Let p be the number of all statements in $\mathcal{T}$, and $\mathcal{H}_{con} = [s_1, s_2, ..., s_p]$ be a schedule of the concurrent transactions in $\mathcal{T}$. We use $\mathcal{H}_{ser} = [T_{x_1}, T_{x_2}, ..., T_{x_n}]$ to denote a serial schedule of $\mathcal{H}_{con}$, in which, $\forall T_{x_i}, T_{x_i} \in \mathcal{T}$.

Definition 1. Transaction-level WSS (`tx-WSS` for short). A schedule $\mathcal{H}_{con}$ of concurrent transactions produces the same final database state after executing $\mathcal{H}_{con}$ as that produced by a corresponding serial schedule $\mathcal{H}_{ser}$ of the same transactions. Otherwise, $\mathcal{H}_{con}$ triggers a `tx-WSS` violation.

In a serial schedule $\mathcal{H}_{ser}$ of concurrent transactions, no involved transactions overlap. The execution of a committed transaction T in $\mathcal{H}_{ser}$ should be equivalent to executing each SQL statement in T as an ACS transaction and ignoring T's transaction control statements, e.g., `BEGIN` and `COMMIT`. For an aborted transaction T in $\mathcal{H}_{ser}$, its execution should be equivalent to ignoring all its statements.

Based on this observation, we further propose a variant of `tx-WSS`, named statement-level `WSS`. Let $\mathcal{H}_{ser}^{stmt}$ be the simplified serial schedule of $\mathcal{H}_{ser}$, in which we remove all the statements in the aborted transactions and transaction control statements in the committed transactions in $\mathcal{H}_{ser}$. We further schedule each statement in $\mathcal{H}_{ser}^{stmt}$ as an ACS transaction, i.e., an autocommit statement.

Definition 2. Statement-level WSS (`stmt-WSS` for short). A schedule $\mathcal{H}_{con}$ of concurrent transactions produces the same final database state after executing $\mathcal{H}_{con}$ as that produced by a corresponding simplified serial schedule $\mathcal{H}_{ser}^{stmt}$ of the same transactions. Otherwise, $\mathcal{H}_{con}$ triggers a `stmt-WSS` violation.

[3]If a write operation relies on the values read in earlier read operations in its transaction (e.g., BEGIN; SELECT @x := c1 FROM t; UPDATE t SET c2 = @x; COMMIT;), concurrent transactions may lead to transaction concurrency problems [29, 38, 54, 56, 67, 74], which are not caused by incorrect transaction processing mechanisms in DBMSs.

The concurrent schedule in Figure 1a produces a different database state from the correct database state produced by the serial schedule $T_2 \rightarrow T_1$ in Figure 1b. Thus, the schedule in Figure 1a triggers a `tx-WSS` violation. The simplified serial schedule (i.e., executing s_{22}, s_{23}, s_{12} and s_{13} as four ACS transactions) can also produce the correct database state. Thus, the schedule in Figure 1a also violates `stmt-WSS`.

Both `tx-WSS` and `stmt-WSS` are necessary for detecting `WSS` violations due to two reasons. (1) If a transaction involves special transaction operations (e.g., `SAVEPOINT` and `ROLLBACK TO` in Listing 2), which cannot be executed outside an explicit transaction, we must keep its transaction control statements (e.g., `BEGIN` and `COMMIT`), and it cannot be analyzed using `stmt-WSS`. (2) If each transaction $T_i \in \mathcal{T}$ contains only one autocommit statement (i.e., $m = 1$), `stmt-WSS` is equivalent to `tx-WSS`. But, for an explicit transaction containing multiple DQL / DML statements, `stmt-WSS` is not equivalent to `tx-WSS`, because DBMSs may process multiple statements in a transaction differently from how they would handle those statements as separate autocommit statements. For example, in Listing 1, executing T_1 as a complete transaction under `tx-WSS` and executing the two DML statements (Line 4 and Line 5) as two autocommit statements under `stmt-WSS` can produce different database states.

Similar to `WSS`, final-state serializability [70] requires that a schedule $\mathcal{H}_{con}$ of concurrent transactions is final-state serializable if there exists a serial schedule of the same transactions that produces the same final database state as $\mathcal{H}_{con}$. Both `WSS` and final-state serializability require that a concurrent schedule produces the same final database state as that of certain serial schedule. However, `WSS` further requires that the serial schedule should satisfy some constraints, e.g., following the *First Commit/Rollback First Scheduled* pattern in Section 3.3. Therefore, `WSS` is stricter than final-state serializability. For example, Listing 4 shows a bug-triggering concurrent schedule that satisfies final-state serializability (since the serial schedule under final-state serializability is $T_2 \rightarrow T_1$), while it does not satisfy `WSS` (since the serial schedule under `WSS` is $T_1 \rightarrow T_2$).

3 EMPIRICAL STUDY ON WSS VIOLATIONS

To better understand `WSS` violations, we conduct an empirical study on real-world critical `txBugs` and their `WSS` violations, and try to answer the following three research questions:

- **RQ1 (Detection capability):** How effectively can `WSS` expose critical `txBugs`?
- **RQ2 (Serial patterns):** To expose `WSS` violations, what serial patterns can be used to infer the serial schedules under `WSS`?

Table 1: Target DBMSs and Collected Critical txBugs

DBMS	DB-Engines Ranking	GitHub Stars	Isolation Levels	Concurrency Control	Type	All txBugs	Critical txBugs	WSS Violations		
								Total	tx-WSS	stmt-WSS
MySQL	2	11.4K	RU, RC, RR, SER	Pessimistic	Traditional	33	2	2	1	2
PostgreSQL	4	17.6K	RC, RR, SER	Pessimistic, Optimistic	Traditional	6	1	1	1	1
SQLite	10	7.8K	RU, SER	Pessimistic	Embedded	6	2	2	0	2
MariaDB	13	6.1K	RU, RC, RR, SER	Pessimistic	Traditional	24	5	3	1	3
CockroachDB	68	30.9K	SER	Optimistic	NewSQL	9	2	2	0	2
TiDB	73	38.5K	RC, RR	Pessimistic, Optimistic	NewSQL	62	23	22	12	22
Total	-	-	-	-	-	**140**	**35**	**32**	**15**	**32**

- **RQ3 (Triggering conditions):** How are WSS violations triggered? Can WSS violations be triggered deterministically?

3.1 Study Methodology

Target DBMSs. We choose six diverse and widely-used open-source relational DBMSs, including three traditional DBMSs (MySQL, PostgreSQL and MariaDB), one embedded DBMS (SQLite), and two NewSQL distributed DBMSs (CockroachDB and TiDB). Table 1 shows their DB-Engines Ranking [5] and Github stars [7]. We can see that these DBMSs are all popular DBMSs. These DBMSs adopt various concurrency control modes, e.g., pessimistic and optimistic transaction modes [50, 73], and support diverse isolation levels [1, 19, 20, 25], e.g., `Read Uncommitted` (RU), `Read Committed` (RC), `Repeatable Read` (RR) and `Serializable` (SER) in MySQL.

Collecting critical txBugs. Our target DBMSs usually contain a large number of issues, e.g., 17,643 issues in TiDB [17]. Investigating these bugs is a time-consuming and daunting task. Thus, we start our study from an existing transaction bug dataset [37]. This dataset contains 140 txBugs collected from the above six DBMSs, which were reported from January 2018 to December 2022. We choose this bug dataset as the base of our study for the following two reasons. First, this dataset covers all our target DBMSs. Second, this dataset contains concise bug descriptions including transaction test cases, bug manifestations, root causes, bug impacts and fixes.

We identify critical txBugs from the bug dataset [37] by checking whether a txBug can cause concurrent transactions to produce an incorrect database state. Table 1 shows the detailed results. We finally collect 35 critical txBugs, and use them as our study subject.

Analyzing critical txBugs and WSS violations. We investigate whether a critical txBug *txbug* can be exposed by WSS by the following steps. First, we manually analyze *txbug*'s transaction test case to infer its *expected* final database state by following the submitted order and execution order of the SQL statements specified by DBMS developers. Next, we enumerate all possible serial schedules for the transactions involved in *txbug*'s test case under tx-WSS and stmt-WSS, respectively. If any serial schedule can produce the *expected* database state, which is different from the database state caused by *txbug*'s test case, we consider *txbug* as a WSS violation. Finally, we collect 32 WSS violations from the 35 critical txBugs.

Threats to validity. First, we may introduce human errors when studying these txBugs and WSS violations. To mitigate this threat, three authors have independently investigated all txBugs and WSS violations, and reached a consensus for each txBug and WSS violation. Second, the reproducibility is a commonly recognized limitation for empirical studies. To make our study reproducible, we have made our study results publicly available for validation.

3.2 txBug Detection Capability of WSS

Among the 35 studied critical txBugs, 32 (91.4%) txBugs cause silent failures, e.g., not crashing DBMSs, which can easily go unnoticed by DBMS developers. Without deep understanding of the transaction semantics in these txBugs' test cases, it is hard to determine whether their produced database states are correct.

```
1. /*init*/ CREATE TABLE t(c1 INT PRIMARY KEY) PARTITION BY RANGE
     (c1) (PARTITION p0 VALUES LESS THAN (10), PARTITION p1
     VALUES LESS THAN MAXVALUE);
2. /*init*/ INSERT INTO t(c1) VALUES (1);
3. /*T1*/ BEGIN;
4. /*T1*/ INSERT INTO t(c1) VALUES (10);
5. /*T1*/ UPDATE t SET c1=c1+10 WHERE c1 IN (1,11);
6. /*T1*/ COMMIT;
7. /*T2*/ SELECT * FROM t ORDER BY c1;
    -- Actual database state:   {10,21} ✘
    -- Expected database state: {10,11} ✔
```

Listing 1: A test case triggers critical txBug TiDB#19585.

We find that, in 15 (42.9%) critical txBugs, their transaction test cases violate tx-WSS, while in 32 (91.4%) critical txBugs, their transaction test cases violate stmt-WSS. Note that all the studied txBugs exposed by tx-WSS are also exposed by stmt-WSS. In 17 txBugs, their transaction test cases violate stmt-WSS, but do not violate tx-WSS, since these txBugs can still occur as long as their transaction control statements exist. Listing 1 shows txBug TiDB#19585 exposed by only stmt-WSS, which is caused by forgetting to deal with some corner cases for DirtyTable. In Listing 1, the initial table t contains a row ($row1$) with $c1 = 1$ (Line 1-2). T_1 first inserts a new row ($row2$) with $c1 = 10$ (Line 4), and then increases $row1$ by 10, which satisfies $c1$ is 1 or 11 (Line 5). Thus, the correct database state is {10, 11}. Executing Listing 1 by following its order produces an incorrect database state {10, 21}. Under tx-WSS, we obtain the database state that is the same as that of the original schedule, while under stmt-WSS, we execute the statements at Line 4, 5, 7 as ACS transactions one by one, and obtain {10, 11} for the database state.

```
1. /*init*/ CREATE TABLE t(c1 TEXT);
2. /*T1*/ BEGIN;
3. /*T1*/ INSERT INTO t(c1) VALUES (REPEAT('a',20000));
4. /*T1*/ SAVEPOINT sp;
5. /*T1*/ INSERT INTO t(c1) VALUES (REPEAT('a',20000));
6. /*T1*/ ROLLBACK TO sp;
7. /*T1*/ COMMIT;
    -- Actual database state:   A crash occurs ✘
    -- Expected database state: {aaaa...aaa} ✔
```

Listing 2: A test case triggers critical txBug MariaDB#14868.

In the remaining 3 txBugs, their transaction test cases do not violate WSS. These 3 txBugs are related to special transaction operations, which cannot be handled by WSS. For example, in txBug MariaDB#14868 in Listing 2, the test case does not violate tx-WSS. Further, we cannot apply its serial schedule for stmt-WSS, since the test case contains SAVEPOINT and ROLLBACK TO statements, which cannot be executed without an explicit transaction.

Finding 1: *In most (91.4%) critical* txBug*s, their transaction test cases violate* WSS*. This indicates that we can use* WSS *as an effective test oracle to detect critical* txBug*s.*

3.3 Serial Patterns under WSS

In our empirical study, we analyze the expected database state mentioned by a transaction test case to infer its corresponding serial schedule under WSS. However, for WSS violation detection, we do not know a transaction test case's expected database state in advance. We are interested in what serial patterns can be used to infer the serial schedules under WSS without leveraging the expected database states of transaction test cases. In this way, we can efficiently detect unknown WSS violations.

By using a serial pattern to infer a transaction test case's serial schedule under WSS, we expect to achieve the following target: For a WSS violation *wssv*, its transaction test case violates WSS before *wssv* is fixed, and its transaction test case does not violate WSS after *wssv* is fixed. For example, in Figure 1b, the serial schedule should be $T_2 \rightarrow T_1$, rather than $T_1 \rightarrow T_2$. If we choose $T_1 \rightarrow T_2$ as the serial schedule, the concurrent schedule in Figure 1a can still violate WSS after the violation is fixed. This is not what we expect.

Through analyzing the transaction test cases in the 32 WSS violations, we observe a simple serial pattern to infer serial schedules under WSS: *First Commit/Rollback First Scheduled* (*FCRFS* for short). If transaction T_1 is committed or aborted before another transaction T_2, T_1 should be scheduled before T_2 in the serial schedule. We find that, by using the FCRFS serial pattern to infer serial schedules, we can detect all the 32 WSS violations. In Section 4, we further formally analyze the correctness of the FCRFS serial pattern.

Finding 2: *By using a serial pattern of First Commit/Rollback First Scheduled to infer serial schedules under* WSS*, we can detect all our studied* WSS *violations without knowing their expected database states.*

3.4 Triggering Conditions

Similar to the empirical study in [37], we study the initial databases and transactions for the 32 WSS violations. Then, we discuss whether these WSS violations can be deterministically triggered.

Initial databases. Triggering a WSS violation usually requires its transactions to be executed on a specific database. Generating a database includes creating initial tables through CREATE TABLE statements and inserting initial data through INSERT statements at the initialization stage in the transaction test cases. Figure 2a and Figure 2b show that all WSS violations require no more than two initial tables, and almost all (97.0%) initial tables contain no more than 5 rows of initial data.

We find that two thirds (71.9%) of WSS violations require specific schema properties on the initial tables, e.g., keys, indexes and column constraints. Specifically, 22 (68.8%) WSS violations require key constraints (e.g., primary key and unique key), 4 (12.5%) WSS violations require index settings, and 6 (18.8%) WSS violations require column constraints (e.g., NOT NULL).

Finding 3: *Almost all our studied* WSS *violations require small initial databases, and two thirds (71.9%) of WSS violations require specific schema properties.*

Transactions. WSS violations are triggered by executing explicit and ACS transactions involved in their test cases. All WSS violations require one to three explicit transactions, and zero to four ACS transactions. Figure 2c and Figure 2d show the distribution of transactions for WSS violations and SQL statements in the transactions for WSS violations, respectively. All WSS violations require no more than 5 transactions, and almost all (96.0%) transactions contain no more than 5 SQL statements. Note that many WSS violations involve SQL statements with complex features, e.g, ALTER (25.0%), DELETE (21.9%), SET (9.4%), REPLACE (6.3%) and ADMIN (6.3%).

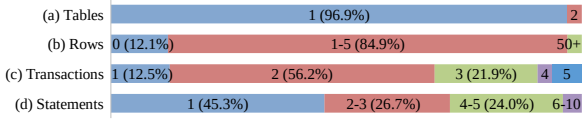

Figure 2: Triggering conditions for WSS violations.

Finding 4: *Almost all our studied* WSS *violations can be triggered by small transaction test cases, e.g., no more than 5 transactions and no more than 5 SQL statements in each transaction.*

Bug determinism. Transactions are executed concurrently in DBMSs, which means their executions are usually non-deterministic. However, we surprisingly find that 29 (90.6%) WSS violations can be triggered by deterministically executing the statements in their transaction test cases in a certain order, e.g., Figure 1a. The remaining 3 (9.4%) WSS violations cannot be triggered deterministically due to internal non-determinism in DBMSs. Triggering these non-deterministic WSS violations requires repeatedly executing their SQL statements in the test cases, or additional deterministic control on DBMSs' internal concurrency. For example, MySQL#99174 needs to inject a crash at a specific internal state in MySQL.

Finding 5: *90.6% of our studied* WSS *violations can be triggered by deterministically executing their statements in a certain order.*

4 CORRECTNESS ANALYSIS OF WSS UNDER THE FCRFS SERIAL PATTERN

In Section 3.3, we find that by using a serial pattern of *First Commit/Rollback First Scheduled* (*FCRFS*) to infer serial schedules under WSS, we can detect all our studied WSS violations. In this section, we further formally analyze the correctness of WSS by using the FCRFS serial pattern.

Given a schedule $\mathcal{H}$ of concurrent transactions $\mathcal{T} = \{T_1, T_2, ..., T_n\}$, we define a happens-before relation $\prec_{\mathcal{H}}$ as a partial order among statements and transactions in $\mathcal{H}$. We denote statement s_1 happens before s_2 in $\mathcal{H}$ as $s_1 \prec_{\mathcal{H}} s_2$, and transaction T_1 happens before T_2 in $\mathcal{H}$ as $T_1 \prec_{\mathcal{H}} T_2$.

We use $\mathcal{H}_{fcrfs} = [T_{x_1}, T_{x_2}, ..., T_{x_n}]$ to denote $\mathcal{H}_{con}$'s FCRFS serial schedule under tx-WSS. In $\mathcal{H}_{fcrfs}$, $\forall T_{x_i}, T_{x_j} \in \mathcal{T}$, if $T_{x_i}.end \prec_{\mathcal{H}_{con}} T_{x_j}.end$, then $T_{x_i} \prec_{\mathcal{H}_{fcrfs}} T_{x_j}$. Here, $T_{x_i}.end$ and $T_{x_j}.end$ refer to the transaction ending statements in T_{x_i} and T_{x_j}, which can be COMMIT, ROLLBACK, or the autocommit statement in an ACS transaction.

Assumption 1. A DBMS can precisely identify conflicts [19, 20] among write operations (e.g., INSERT, UPDATE, and DELETE) in a transaction test case, and correctly process transactions (e.g., block or abort transactions) according to these conflicts. For example, statement s_{13} and s_{22} in Figure 1a should be identified as conflicts, since the new inserted value 2.0 satisfies $c1 < 5.0$. Note that not all isolation levels in some DBMSs can support this assumption. We will discuss this more in Section 6.4.

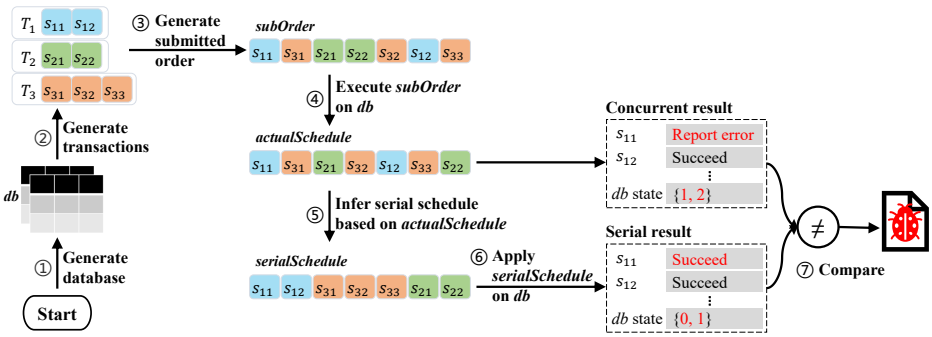

Figure 3: The workflow of `WriteCheck`.

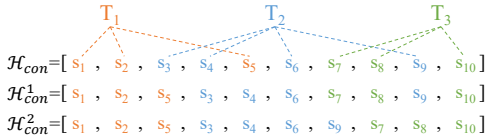

Figure 4: An illustrative example for proving Theorem 1.

Theorem 1. If a DBMS satisfies Assumption 1, a schedule $\mathcal{H}_{con}$ of concurrent transactions produces the same final database state as that produced by the FCRFS serial schedule $\mathcal{H}_{fcrfs}$ of the same transactions.

Let db_{init} be the initial database state, $db(\mathcal{H}_{con})$ be the database state after applying $\mathcal{H}_{con}$ on db_{init}, and $db(\mathcal{H}_{fcrfs})$ be the database state after applying $\mathcal{H}_{fcrfs}$ on db_{init}. We prove that $db(\mathcal{H}_{con}) = db(\mathcal{H}_{fcrfs})$, if a DBMS satisfies Assumption 1.

Proof. (1) If there is only one transaction in $\mathcal{H}_{con}$, i.e., $n = 1$, then $\mathcal{H}_{con} = \mathcal{H}_{fcrfs}$ and $db(\mathcal{H}_{con}) = db(\mathcal{H}_{fcrfs})$.

(2) When $n > 1$, for the first completed transaction T_{x_1} in $\mathcal{H}_{con}$, we can infer that, for each statement s_i that satisfies $s_i \notin T_{x_1}$ and $s_i \prec_{\mathcal{H}_{con}} T_{x_1}.end$, s_i does not conflict with T_{x_1}. Otherwise, T_{x_1} cannot complete before s_i's transaction completes in $\mathcal{H}_{con}$. Therefore, in $\mathcal{H}_{con}$, we can move each statement such s_i behind $T_{x_1}.end$, and form a new schedule $\mathcal{H}^1_{con}$, which can produce the same database state as $\mathcal{H}_{con}$, i.e., $db(\mathcal{H}^1_{con}) = db(\mathcal{H}_{con})$. Thus, all statements in T_{x_1} are located at the head of $\mathcal{H}^1_{con}$. We continue the above analysis process for the remaining transactions in $\mathcal{H}^1_{con}$, and generate $\mathcal{H}^2_{con}$, ...,$\mathcal{H}^{n-1}_{con}$, in which $db(\mathcal{H}_{con}) = db(\mathcal{H}^1_{con}) = db(\mathcal{H}^2_{con}) = ... = db(\mathcal{H}^{n-1}_{con})$. Based on the above analysis, $db(\mathcal{H}_{fcrfs}) = db(\mathcal{H}^{n-1}_{con})$. Thus, $db(\mathcal{H}_{con}) = db(\mathcal{H}_{fcrfs})$.

Figure 4 shows an example for the above proof process. Three concurrent transactions T_1, T_2 and T_3 generate a schedule $\mathcal{H}_{con}$, in which $T_1.end$ (i.e., s_5) $\prec_{\mathcal{H}_{con}}$ $T_2.end$ (i.e., s_9) $\prec_{\mathcal{H}_{con}}$ $T_3.end$ (i.e., s_{10}). Since T_1 has completed at s_5, s_3 and s_4 should not conflict with T_1. So, we can move s_3 and s_4 behind s_5, and form a schedule $\mathcal{H}^1_{con}$, which satisfies $db(\mathcal{H}^1_{con}) = db(\mathcal{H}_{con})$. Similarly, we can obtain a schedule $\mathcal{H}^2_{con}$, which satisfies $db(\mathcal{H}^1_{con}) = db(\mathcal{H}^2_{con})$. $\mathcal{H}^2_{con}$ is the same as $\mathcal{H}_{fcrfs}$. Therefore, $db(\mathcal{H}_{fcrfs}) = db(\mathcal{H}_{con})$.

We further define the simplified serial schedule under `stmt-WSS` for $\mathcal{H}_{fcrfs}$ as $\mathcal{H}^{stmt}_{fcrfs}$, which removes all the aborted transactions and transaction control statements in the committed transactions (e.g., `BEGIN` and `COMMIT`) from $\mathcal{H}_{fcrfs}$.

Theorem 2. If a DBMS satisfies Assumption 1, a schedule $\mathcal{H}_{con}$ of concurrent transactions produces the same final database state as that produced by the simplified serial schedule $\mathcal{H}^{stmt}_{fcrfs}$ of the same transactions.

Proof. Since the aborted transactions and transaction control statements in the committed transactions do not affect the database state, then $db(\mathcal{H}_{fcrfs}) = db(\mathcal{H}^{stmt}_{fcrfs})$. As we have proved $db(\mathcal{H}_{con}) = db(\mathcal{H}_{fcrfs})$, then $db(\mathcal{H}_{con}) = db(\mathcal{H}^{stmt}_{fcrfs})$.

5 DETECTING `WSS` VIOLATIONS

We propose `WriteCheck`, a simple and general approach for automatically detecting `WSS` violations in DBMSs. As shown in Figure 3, we first generate an initial database with some data (①). We then generate a group of transactions based on the generated database (②), and further generate a submitted order for the statements in the generated transactions (③). Then, we submit the transaction statements to the target DBMS by following the submitted order (④). During execution, we obtain the concurrent schedule and execution results of the transactions, including the execution results of write operations and the final database state. Based on the concurrent schedule, we infer the corresponding serial schedules under `tx-WSS` and `stmt-WSS` by using the FCRFS serial pattern (⑤). We then apply the two serial schedules on the same initial database to obtain the serial execution results (⑥). Finally, we compare the execution results between the concurrent schedule and the serial schedules (⑦). Any inconsistency indicates a `WSS` violation.

We stressfully test a target DBMS by iteratively executing the above steps. If we do not detect a `WSS` violation in one iteration, we can continue generating a new submitted order for the current transactions (③), or generating a new group of transactions based on the current database (②), or generating a new database (①).

5.1 Generating Databases and SQL Statements

Many approaches have been proposed for database generation [27, 30, 40, 41, 48] and SQL statement generation [8, 16, 31, 61, 78]. We utilize SQLancer [58–60] to generate initial databases and individual SQL statements, and extend SQLancer to generate transactions and transaction test cases.

Generating databases. We use `CREATE TABLE` statements to create tables in a database. We randomly assign a data type and add data constraints (e.g., `PRIMARY KEY` and `NOT NULL`) for each column. Note that we do not generate variable data types, whose data are randomly generated, e.g., `SERIAL` in CockroachDB. We further randomly add indexes to the selected columns, and insert rows into each table. Based on Finding 3, we generate small databases.

Generating SQL statements. Our target DBMSs support slightly different SQL dialects. For example, PostgreSQL does not support `REPLACE`, and only CockroachDB supports `UPSERT`. To generate

valid SQL statements, we construct SQL statements based on the respective SQL grammars supported by target DBMSs.

We can support various data query and manipulation statements (i.e., DQL and DML statements), and transaction control statements, e.g., `SELECT`, `INSERT`, `REPLACE`, `UPSERT`, `UPDATE` and `DELETE`. We also support almost all SQL features in these DQL and DML statements, e.g., complex predicates, subqueries, join and dialects in different DBMSs. Our approach cannot support few SQL features, i.e., non-deterministic functions (e.g., `RAND()` and `UUID()`), and variable declaration and usage (e.g., $@x$ in MySQL).

5.2 Generating Transactions

We name the transactions in a transaction test case as a transaction group. For each transaction tx in a transaction group, we first randomly choose a transaction type for tx, i.e., explicit or ACS transaction. If tx is an explicit transaction, we utilize the approach in Section 5.1 to randomly generate at most *maxStmt* SQL statements for accessing data, append a `BEGIN` statement at the beginning, and a `COMMIT` or `ROLLBACK` statement (which is randomly chosen) at the end. If tx is an ACS transaction, we randomly generate a SQL statement, which forms tx. We generate at most *maxTxn* transactions in a transaction group. For effective `WSS` violation detection, we require that a transaction group contains at least one explicit transaction, and at least one transaction contains write operations.

According to Finding 4, most `WSS` violations can be triggered by relatively small transaction test cases. So we set both *maxTxn* and *maxStmt* as 5 by default. For a transaction group, we further randomly choose an isolation level and a concurrency control mode supported by the tested DBMS.

Generating submitted orders. Finding 5 shows that most `WSS` violations can be triggered deterministically by executing their involved SQL statements in a certain order. Thus, we generate a submitted order for the SQL statements in a transaction group. DBMSs will process these statements in the submitted order and execute them in a deterministic order.

To test diverse transaction scenarios as soon as possible, we randomly generate a submitted order, rather than enumerating all possible submitted orders for a transaction group. Specifically, we randomly choose a transaction tx in the transaction group each time, and append tx's first unchosen statement to the submitted order, until all statements in the transaction group have been chosen.

5.3 Obtaining Actual Concurrent Schedules

When we submit the statements one by one to a tested DBMS by following a given submitted order, the DBMS may encounter various situations, e.g., a statement is blocked, a deadlock is raised, and some blocked statements are resumed. Thus, the actual concurrent schedule of a transaction test case is usually different from the submitted order of the statements in the involved transactions. To tackle this problem, we design a transaction execution protocol, which can obtain a transaction test case's actual schedule. This protocol can support multiple transactions and is general to different DBMSs and concurrency control modes. DBMSs usually adopt different transaction processing mechanisms for raised errors, deadlocks, etc. We will further explain how to handle them in Section 5.3.2.

5.3.1 Transaction Execution Protocol. As illustrated in Algorithm 1, we submit the statements one by one to the tested DBMS in a given

Algorithm 1: Transaction execution protocol

```
   Input: subOrder
   Output: actualSchedule
1  while subOrder.hasStmtToSchedule() do
2      for i ← 1; i ≤ subOrder.length; i++ do
3          stmt ← subOrder[i]
4          curTx ← stmt.transaction
5          if stmt.submitted then
6              continue
7          if curTx.aborted | curTx.blocked then
8              continue
9          execState ← curTx.submit(stmt)
10         stmt.submitted ← True
11         if stmt.execState.isBlocked() then
12             curTx.blocked ← True
13             continue
14         actualSchedule.add(stmt)
15         if stmt.execState.shouldAbort() then
16             curTx.aborted ← True
17             handleAbort(actualSchedule, curTX)
18         rStmts ← getResumedStmts()
19         foreach rStmt ∈ rStmts do
20             rTx ← rStmt.transaction
21             rTx.blocked ← False
22             actualSchedule.add(rStmt)
23             if rStmt.execState.shouldAbort() then
24                 rTx.aborted ← True
25                 handleAbort(actualSchedule, rTx)
26         if rStmts ≠ ∅ then
27             break
```

submitted order (Line 2-3). For each statement *stmt*, we start a new thread to execute it independently, obtain its execution state *stmt.execState*, and mark *stmt* as submitted (Line 9-10). If *stmt* has already been submitted, we avoid executing it again (Line 5-6).

If *stmt*'s transaction *curTx* has been aborted or blocked, we also skip submitting *stmt* (Line 7-8). Based on *stmt*'s execution state *execState*, we can decide whether *curTx* should be blocked or aborted. If *stmt* does not return its execution results within *maxWait* seconds (2 seconds by default), we consider that *stmt* is blocked (e.g., a conflict occurs). Then we mark *curTx* as blocked and continue to schedule next statement in *subOrder* (Line 11-13). If *stmt* is not blocked, we append *stmt* to *actualSchedule* and record *stmt*'s execution results (i.e., whether *stmt* succeeds or not) (Line 14). If the tested DBMS alerts to abort a transaction (e.g., a deadlock occurs), we mark *curTx* as aborted and abort *curTx* (Line 15-17).

After executing *stmt*, we check whether any blocked statements have been resumed by the tested DBMS (Line 18). There are two scenarios in which a blocked statement can be resumed. First, *stmt* is a transaction end statement, i.e., `COMMIT` or `ROLLBACK`. Second, *stmt*'s transaction *curTx* is aborted by the tested DBMS. For a resumed statement *rStmt*, we mark its transaction as unblocked, append it to *actualSchedule* and record its execution results (Line 19-22). Similarly, if *rStmt* causes the DBMS to abort its transaction *rTx*, we mark *rTx* as aborted and abort *rTx* (Line 23-25). If any blocked statements are resumed by the DBMS, we will scan *subOrder* from the beginning, and execute not-submitted statements in *subOrder* sequentially (Line 26-27 and Line 1).

5.3.2 Handling Transaction Aborts in Different DBMSs. Although Algorithm 1 is general to our target DBMSs, different DBMSs may adopt slightly different transaction processing mechanisms to abort transactions. If we do not adjust the abort handling for different

DBMSs, WriteCheck would possibly report false positives. Thus, we design specific approaches to identify whether we need to abort a transaction (Line 15 and 23), and how to process transaction aborts (Line 17 and 25) for different DBMSs. We elaborate them as follows.

Deadlock. When a deadlock occurs under the pessimistic transaction mode, MySQL, MariaDB and TiDB will report a deadlock error in one of the involved transactions. However, SQLite does not report a deadlock error when a deadlock occurs. We adopt a simple strategy to detect potential deadlocks in SQLite: If a transaction $tx1$ has been blocked and we detect that a new transaction $tx2$ is blocked, we conservatively consider that a deadlock occurs in $tx2$.

When a deadlock occurs, MySQL and TiDB will automatically abort the transaction that reports a deadlock. While for SQLite and MariaDB, we need to submit a ROLLBACK statement to abort the explicit transaction that reports a deadlock.

Write-write conflict. Under the optimistic transaction mode, when committing a transaction, TiDB checks write-write conflict. If write-write conflict occurs, TiDB aborts the transaction automatically. Thus, we remove the COMMIT statement from *actualSchedule*.

Error. In PostgreSQL and CockroachDB, an error reported at a transaction will interrupt this transaction and causes subsequent statements in the transaction to report errors until the transaction is aborted. In this case, we submit a ROLLBACK statement to the DBMS to abort the transaction.

After aborting an explicit transaction, we need to add a ROLLBACK statement into *actualSchedule*. For a failed ACS transaction, we only mark it as aborted.

5.4 Identifying WSS Violations

Finding 2 shows that WSS violations follow the FCRFS serial pattern. To infer the serial schedule under tx-WSS, we sequentially scan the actual concurrent schedule of a transaction test case and build a transaction list by appending transactions upon encountering a COMMIT or ROLLBACK. For ACS transactions, we append them directly. For the serial schedule under stmt-WSS, we create a statement list by scanning the transaction list, ignoring aborted transactions and transaction control statements from committed transactions, and appending the remaining statements from the committed transactions and the committed ACS transactions.

Serial schedules for both tx-WSS and stmt-WSS are applied on the same initial database as that in the actual schedule in Section 5.3. During each serial schedule, we record each statement *stmt*'s execution result (i.e., whether *stmt* succeeds or not). Then, we retrieve data in each table in the database as the final database state.

To detect tx-WSS and stmt-WSS violations, we compare the final database states in actual schedule with those in transaction-level and statement-level serial schedules, respectively. Any inconsistencies indicate WSS violations.

In DBMSs, when a write operation succeeds, it usually causes changes to the database state. When the write operation reports an error and fails to execute, it will not affect the database state. Therefore, by comparing whether the same write operation in actual concurrent schedule and a serial schedule succeeds or not, we can infer inconsistent database states and detect WSS violations.

Identifying incorrect query results of read operations at the Serializable isolation level. Although we focus on detecting WSS violations in this paper, our approach can easily identify incorrect query results of read operations at the Serializable isolation level. At this isolation level, the execution of concurrent transactions should be equivalent to their execution in certain serial schedule. The serial schedule of concurrent transactions under WSS can be treated as their intended serial schedule. Thus, at this isolation level, we compare the query results of read operations in each transaction between the concurrent schedule and the corresponding serial schedule under WSS. Any inconsistencies indicate a txBug that causes wrong query results for the Serializable isolation level[4]. Note that such txBugs may not violate WSS.

6 EVALUATION

We evaluate WriteCheck on six production-level DBMSs, and address the following two research questions.

- **RQ4:** How effective is WriteCheck in detecting WSS violations in real-world DBMSs?
- **RQ5:** How does WriteCheck compare against existing approaches?

6.1 Experimental Methodology

Target DBMSs. We evaluate WriteCheck on the six DBMSs from our empirical study. We test the latest versions of these DBMSs when we start our work, i.e., MySQL 8.0.32, PostgreSQL 15.2, SQLite 3.36.0, MariaDB 10.11.2, CockroachDB 22.2.5, and TiDB 6.6.0.

Testing setup. We perform our experiment on Ubuntu-20.04 with 8 CPU cores and 32 GB of RAM. We build a Docker container for MySQL, PostgreSQL and MariaDB, respectively, and create an instance in it. We deploy TiDB with one TiDB instance, one TiKV instance and one PD instance. We deploy CockroachDB with a local cluster containing a single node. We embed SQLite in WriteCheck.

Testing methodology. We test the DBMSs for 5 rounds, and each round takes around 24 hours. Once WriteCheck detects a WSS violation, it generates a report, including the transaction test case and the serial schedule under WSS. The transaction test case contains database creation statements, transactions, the submitted order, the concurrency control mode and the isolation level. With this report, we can reproduce and analyze the WSS violation.

In each round, we iteratively and continuously run WriteCheck, and collect all the WSS violations it detects. For a WSS violation reported by WriteCheck, we manually simplify the transaction test case by removing the statements and SQL elements that are irrelevant to the violation, and obtain its simplest bug-triggering transaction test case. We further identify its key bug-triggering SQL features in the simplest bug-triggering test case. If the related bug-triggering SQL features in the simplest test case are removed or replaced, the corresponding bug will not be triggered.

We then apply the following strategies to identify duplicated WSS violations. First, if two WSS violations share the same bug-triggering features in their simplest transaction test cases, we consider them as duplicate. Second, if executing a test case at multiple isolation levels results in the same inconsistent database states, we consider such WSS violations as duplicate. After removing duplicated WSS violations, we report the remaining WSS violations to DBMS developers and wait for feedbacks from developers.

[4]We cannot detect txBugs due to the incorrect results of read operations for the Serializable isolation level (i.e., Serializable Snapshot Isolation) in PostgreSQL [55], in which, the read operations in a reading transaction T_1 may need to be serialized before a write operation in another transaction T_2, even though T_1 commits after T_2.

Table 2: WSS Violations Detected by WriteCheck

DBMS	Isolation Level				Total	txBug		False
	RU	RC	RR	SER	(Unique)	New (Fixed)	Duplicate	Positive
MySQL	8	8	0	0	16 (3)	1 (0)	0	2
PostgreSQL	-	6	10	10	26 (3)	0 (0)	0	3
SQLite	0	-	-	0	0 (0)	0 (0)	0	0
MariaDB	10	10	1	1	22 (5)	3 (1)	0	2
CockroachDB	-	-	-	0	0 (0)	0 (0)	0	0
TiDB	-	18	19	-	37 (11)	7 (1)	2	2
Total	18	42	30	11	101 (22)	11 (2)	2	9

6.2 Overall WSS Violations

In total, WriteCheck has detected 101 WSS violations in the six target DBMSs, as shown in Table 2. WriteCheck has not detected any WSS violations in SQLite and CockroachDB. It is reasonable to observe inconsistent bug detection performance across DBMSs. DBMSs are complex and usually involve different transaction mechanisms and implementations. The numbers of txBugs in different DBMSs may vary greatly. A DBMS that adopts simple transaction mechanisms and implementations (e.g., SQLite) may contain less txBugs, while a DBMS that adopts complex transaction mechanisms and implementations (e.g., TiDB) may contain more txBugs.

From these 101 WSS violations, we identified 22 unique WSS violations, with 6 only detectable by stmt-WSS and 16 by both tx-WSS and stmt-WSS. We submitted these unique WSS violations to the relevant DBMS community and discussed them with DBMS developers. 13 WSS violations have been confirmed as critical txBugs, among which, 11 WSS violations have been confirmed as new critical txBugs (2 txBugs have been fixed by DBMS developers), and 2 WSS violations are considered as duplicate to existing bugs. The remaining 9 WSS violations are considered false positives, which are caused by DBMS design choices or improper DBMS designs.

For the 13 WSS violations confirmed as critical txBugs, 4, 12, 10 and 1 WSS violations are exposed at the Read Uncommitted, Read Committed, Repeatable Read and Serializable isolation levels, respectively. In addition, one WSS violation is exposed under the optimistic transaction mode in TiDB. The other 12 WSS violations are exposed under the pessimistic transaction mode in their corresponding DBMSs. This shows that WriteCheck can detect WSS violations at all isolation levels and concurrency control modes.

We further utilize WriteCheck to detect non-critical txBugs due to read operations at the Serializable isolation level (except the Serializable isolation level in PostgreSQL [55]), as discussed in Section 5.4. However, we have not found such non-critical txBugs yet. We further investigate the txBugs reported by existing approaches [36, 39, 45], and find that these approaches did not report non-critical txBugs due to read operations at the Serializable isolation level, either. We guess that non-critical txBugs rarely occur at the Serializable isolation level in DBMSs.

> WriteCheck *has detected 13 critical* txBug*s, 11 of which have been confirmed as new critical* txBug*s, and 2 critical* txBug*s have been fixed by DBMS developers. This demonstrates the effectiveness of* WriteCheck.

6.3 Interesting Critical txBugs

We further explain more newly detected critical txBugs that have not been detailed previously.

TiDB#39976. Listing 3 shows a critical txBug detected in TiDB. This txBug can be triggered by only one transaction T_1, which first inserts a new value '1' into column $c1$ in table t (Line 3) and then tries to delete the rows satisfying its WHERE condition (Line 4). However, the DELETE statement reports an error and fails to delete value '1'. While in the serial schedule under stmt-WSS (i.e., ignoring Line 2 and 5), the DELETE statement successfully deletes value '1'. Thus, this transaction test case violates stmt-WSS. This txBug is caused by inconsistent evaluation of the WHERE condition in the DELETE statement within and without a transaction.

```
1. /*init*/ CREATE TABLE t(c1 TEXT(5));
2. /*T1*/ BEGIN;
3. /*T1*/ REPLACE INTO t(c1) VALUES ('1');
4. /*T1*/ DELETE FROM t WHERE CAST(TIDB_VERSION() AS DATE) OR c1;
      -- report an error
5. /*T1*/ COMMIT;
    -- Actual database state:   {1} ✘
    -- Serial database state:   { } ✔
```

Listing 3: A test case triggers critical txBug TiDB#39976 at the Read Committed and Repeatable Read isolation levels.

TiDB#42486. Listing 4 shows a critical txBug detected in TiDB at the Repeatable Read isolation level under the optimistic transaction mode. The test case involves two transactions T_1 and T_2. T_1 first inserts a value 2 and commits (Line 5-6). Then T_2 deletes all values in table t and commits (Line 7-8). T_2 should be aborted due to its conflict with T_1, but it successfully commits (Line 8), resulting in a database state with value 2. In the serial schedule, i.e., $T_1 \rightarrow T_2$, the table t would be empty, revealing this WSS violation.

```
1. /*init*/ CREATE TABLE t(c1 INT);
2. /*init*/ INSERT INTO t(c1) VALUES (1);
3. /*T1*/ BEGIN OPTIMISTIC;
4. /*T2*/ BEGIN OPTIMISTIC;
5. /*T1*/ INSERT INTO t(c1) VALUES (2);
6. /*T1*/ COMMIT;
7. /*T2*/ DELETE FROM t;
8. /*T2*/ COMMIT; -- T2 should be aborted, but committed successfully.
    -- Actual database state:          {2}
    -- Serial database state (T1 → T2): { }
```

Listing 4: A test case triggers critical txBug TiDB#42486 under the optimistic transaction mode.

MariaDB#30835. Figure 5a shows a critical txBug detected in MariaDB at the Read Uncommitted and Read Committed isolation levels. In the transaction test case, the initial table has a column $c1$ with a value 1.0. T_1 inserts a value 2.0 into column $c1$ (s_{12}). Then T_2 changes the value 2.0 to 3.0 (s_{22}). We can get the final database state, i.e., {1.0, 2.0}. While the serial schedule ($T_1 \rightarrow T_2$) produces the final database state {1.0, 3.0}. Thus, we detect this txBug. This txBug is caused by the relaxed locks for the UPDATE statement (s_{22}).

Replacing the UPDATE statement s_{22} with a DELETE statement, as shown in Figure 5b, results in both concurrent and serial schedules producing the final database state {1.0}. This occurs because the DELETE statement is blocked until T_1 commits in the concurrent schedule, after which the value 2.0 is deleted. Note that two write operations with the same WHERE condition are expected to have the same transaction semantics. DBMS developers are designing new transaction processing mechanisms to address this issue: *"The reason why the locks are being relaxed for UPDATE is the 'semi-consistent read' that I originally implemented before. Implementing it for DELETE was not considered at that time. I would expect that the lock conflicts would be reduced, or possibly avoided altogether".*

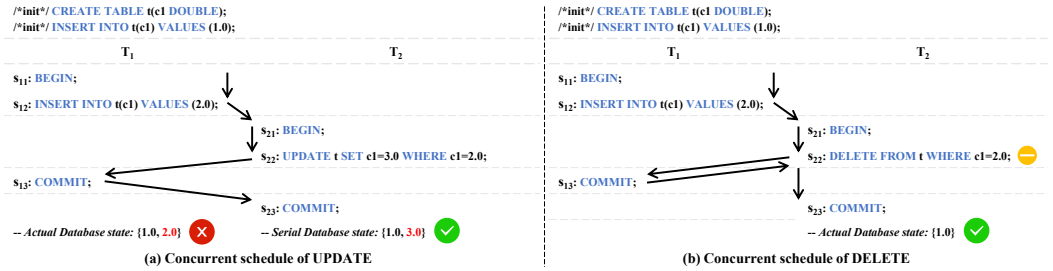

Figure 5: A test case triggers critical txBug MariaDB#30835 at the `Read Uncommitted` and `Read Committed` isolation levels.

6.4 False Positive Analysis

DBMS developers consider 9 WSS violations detected by `WriteCheck` as false positives. We find that these violations stem from improper or inconsistent designs in the target DBMSs, which are permissible in the target DBMSs. We investigate these permissible WSS violations to understand their root causes and consequences.

6.4.1 Root Causes of False Positives. We find that these false positives occur, since they do not satisfy Assumption 1 in Section 4. Specifically, DBMSs do not precisely identify conflicts among write operations, which are caused by semi-consistent read[3, 4, 13] and lack of gap locks[6]. In total, 5 false positives are caused by semi-consistent read and 4 false positives are caused by lack of gap locks.

Listing 5 shows a permissible WSS violation at the `Read Committed` isolation level in MySQL caused by semi-consistent read, which leads to an inconsistent view of the database. In Listing 5, the `UPDATE` statement (Line 4) in T_1 reads the latest committed rows in table t, i.e., (1, '') and (5, ''), and evaluates its `WHERE` condition on each row. The `UPDATE` statement only keeps a lock for each row that matches its condition. Thus, it only locks row (1, '') and modifies it to (5, 'tx1'). Since T_1 has not been committed, the updated row (5, 'tx1') by Line 4 cannot be read by the `UPDATE` statement at Line 6.

```
1. /*init*/ CREATE TABLE t(c1 INT, c2 VARCHAR(5));
2. /*init*/ INSERT INTO t(c1,c2) VALUES (1,''), (5,'');
3. /*T1*/ BEGIN;
4. /*T1*/ UPDATE t SET c1=5, c2='tx1' WHERE c1=1;
5. /*T2*/ BEGIN;
6. /*T2*/ UPDATE t SET c1=1, c2='tx2' WHERE c1=5;
7. /*T1*/ COMMIT;
8. /*T2*/ COMMIT;
   -- Actual database state:          {(5,'tx1'), (1,'tx2')}
   -- Serial database state(T1 → T2): {(1,'tx2'), (1,'tx2')}
```

Listing 5: A permissible WSS violation caused by semi-consistent read at `Read Committed` isolation level in MySQL.

Listing 6 shows a permissible WSS violation at the `Read Committed` isolation level in TiDB caused by lack of gap locks, which lock the gaps between index records. If gap locks are used, the `DELETE` statement prevents T_2 from inserting the value 5 into column $c1$ of table t, as the gaps between all existing values in the range are locked.

```
1. /*init*/ CREATE TABLE t(c1 INT);
2. /*init*/ INSERT INTO t(c1) VALUES (3);
3. /*T1*/ BEGIN;
4. /*T1*/ DELETE FROM t WHERE c1 BETWEEN 1 AND 10;
5. /*T2*/ BEGIN;
6. /*T2*/ INSERT INTO t(c1) VALUES (5);
7. /*T2*/ COMMIT;
8. /*T1*/ UPDATE t SET c1=c1+1;
9. /*T1*/ COMMIT;
   -- Actual database state:          {6}
   -- Serial database state (T2 → T1): { }
```

Listing 6: A permissible WSS violation caused by lack of gap locks at the `Read Committed` isolation level in TiDB.

Gap locks are not used in some isolation levels in some DBMSs. For example, MySQL and MariaDB do not use gap locks at the `Read Committed` and `Read Uncommitted` isolation levels. TiDB does not use gap locks. If gap locks are not used, a transaction test case can violate WSS. For example, in Listing 6, the `INSERT` statement (Line 6) in T_2 at the `Read Committed` isolation level in TiDB can successfully insert the value 5, and does not conflict with the `DELETE` statement at Line 4. The actual database state is {6}, which differs from the database state produced by the serial schedule of $T_2 \rightarrow T_1$.

6.4.2 Consequences Caused by Permissible WSS Violations. Although these WSS violations are considered permissible by DBMS developers, we find that these permissible WSS violations can lead to ambiguous and conflicting transaction semantics for write operations.

A write operation only modifies partial rows that satisfy its `WHERE` condition. Listing 5 shows such a permissible WSS violation at the `Read Committed` isolation level in MySQL. As explained in Section 6.4.1, in this test case, the `UPDATE` in T_1 (Line 4) updates the row (1, ''), and the `UPDATE` in T_2 (Line 6) updates the row (5, ''). After T_1 and T_2 complete, although the row (5, 'tx1') satisfies the `WHERE` condition at Line 6, it is not updated by the write operation. From DBMS users' view, only partial rows that satisfy the `WHERE` condition at Line 6 are updated. This could be confusing.

Different types of write operations in a single DBMS at the same isolation level have inconsistent transaction behaviors. Figure 5 shows such inconsistent transaction behaviors between write operations `UPDATE` and `DELETE` at the `Read Committed` isolation level in MySQL and MariaDB. MariaDB developers have confirmed Figure 5a as a txBug. However, MySQL developers consider it as a permissible WSS violation caused by semi-consistent read. In Figure 5a, the `UPDATE` statement s_{22} is not blocked due to semi-consistent read. However, the `DELETE` statement s_{22} in Figure 5b with the same `WHERE` condition is blocked, since the `DELETE` statement does not use semi-consistent read. We expect that two write operations with the same `WHERE` condition have the same transaction semantics. However, this is not true for MySQL.

Different DBMSs have inconsistent transaction behaviors for the same write operations at the same isolation levels. For example, TiDB utilizes semi-consistent read at the `Repeatable Read` isolation level, while MySQL and MariaDB do not. MySQL and MariaDB utilize gap locks at the `Repeatable Read` isolation level, while PostgreSQL and TiDB do not.

Different isolation levels in a single DBMS have inconsistent transaction behaviors for write operations. For example, MySQL utilizes semi-consistent read at the `Read Uncommitted` and `Read Committed`, not at the `Repeatable Read` and `Serializable` isolation levels.

Table 3: Detection Capability Comparison on the Reported Critical txBugs

DBMS	Reported Critical txBugs			WriteCheck			DT^2			Troc			TxCheck			Elle	Cobra	Emme
	Total	WriteCheck	Existing Approaches	T*	WC	EA	T	WC	EA	T	WC	EA	T	WC	EA			
MySQL	3	1	2	3	1	2	2	1	1	3	1	2	1	0	1	0	0	0
PostgreSQL	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SQLite	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MariaDB	8	3	5	8	3	5	7	2	5	3	1	2	2	0	2	0	0	0
CockroachDB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TiDB	18	9	9	18	9	9	14	7	7	8	5	3	13	4	9	0	0	0
Total	29	13	16	29	13	16	23	10	13	14	7	7	16	4	12	0	0	0

* T (Total), WC (WriteCheck) and EA (Existing Approaches) denote the numbers of critical txBugs detected by the corresponding approach from total reported txBugs (Column 2), txBugs reported by WriteCheck (Column 3) and existing approaches (Column 4), respectively.

6.4.3 Reflection. Existing works on isolation levels have clearly defined the transaction semantics for read operations [1, 19, 20, 25]. For write operations, these isolation levels only require that DBMSs must prohibit dirty write. However, our study shows that different DBMSs usually have own specific transaction semantics of write operations at different isolation levels. Sometimes, these transaction semantics are not clearly specified by DBMSs and even conflicting. To tackle this issue, we urgently need a clear and unified approach to specify transaction semantics for write operations at different isolation levels in different DBMSs.

Different DBMSs usually have their own specific transaction semantics for write operations at different isolation levels, but there is no unified approach to define these transaction semantics.

6.5 Comparison with Existing Approaches

Existing transaction verification (Elle [49], Cobra [66] and Emme [34]) and testing approaches (DT^2 [36], Troc [39] and TxCheck [45]) can also detect critical txBugs. Since all these approaches rely on random testing, it is challenging to compare WriteCheck with them directly about their bug detection capability. Thus, we investigate whether these approaches can conceptually detect the 29 critical txBugs reported by WriteCheck and existing approaches [36, 39, 45] when the corresponding bug-triggering transaction test cases are provided. Then, we perform an end-to-end experimental comparison with the existing transaction testing approaches.

6.5.1 Detection Capability Comparison on Reported Critical txBug*s.* We first collect all the critical txBugs reported by the transaction testing approaches (i.e., WriteCheck, DT^2 [36], Troc [39] and TxCheck [45]). We then check whether these txBugs can be exposed by an approach when the corresponding test cases are provided.

Collecting critical txBugs. We investigate all the confirmed txBug reports from DT^2 [36], Troc [39] and TxCheck [45], and collect 47 txBugs. Among them, 16 txBugs can cause incorrect database states and are classified as critical txBugs. Finally, including the 13 critical txBugs reported by WriteCheck, we collect 29 critical txBugs and obtain their simplest bug-triggering transaction test cases through the simplifying approach in Section 6.1.

Analysis methodology. For each critical txBug *txbug* in our subject, we analyze its simplest bug-triggering transaction test case to evaluate whether a transaction verification or testing approach *app* can expose it. Specifically, if *app* cannot support the bug-triggering SQL features in *txbug*'s test case, or it can support the bug-triggering SQL features but cannot determine that the DBMS behaves wrongly in the test case, *app* cannot detect *txbug*.

Overall comparison result. Table 3 shows the comparison results. For the 29 critical txBugs, WriteCheck detects all of them, while DT^2, Troc and TxCheck detect 23, 14, 16 txBugs, respectively. None of these txBugs can be detected by transaction verification approaches. Notably, among the 13 critical txBugs detected by WriteCheck, existing approaches can identify at most 10, while WriteCheck also detects all 16 critical txBugs reported by these approaches. We explain why existing approaches fail to detect some critical txBugs as follows.

Elle [49], Cobra [66] and Emme [34] verify whether a transaction execution history violates the claimed isolation level. They are built on simple *key-value*-like data models and simple operations (e.g., $read(key)$ and $write(key, value)$). They cannot support non-*key-value*-like data models and complex SQL statements. They detect isolation anomalies by inferring whether there exist cycles in the dependency graphs based on Adya's definition [19, 20], and cannot detect txBugs whose dependency graphs do not contain cycles (e.g., Listing 4). Among the 29 critical txBugs, 26 txBugs involve non-*key-value*-like data models and complex operations, and the remaining 3 txBugs do not involve dependency cycles.

DT^2 [36] detects txBugs by differentially testing the same transaction test cases on multiple DBMSs. Among the 29 critical txBugs, 4 cannot be detected by DT^2 due to DBMS-specific features (e.g., only TiDB supports the optimistic transaction mode), and 2 have identical incorrect behaviors across all tested DBMSs. Troc [39] decouples a pair of transactions into independent statements, and executes them on their own database views under the guidance of a specific isolation level. For the 29 critical txBugs, 15 txBugs involve unsupported bug-triggering SQL features (e.g., more than two transactions) and cannot be detected by Troc. TxCheck [45] designs a SQL-level instrumentation to capture statement dependencies, and generates semantically-equivalent test cases based on statement dependencies. For the 29 critical txBugs, 10 txBugs involve bug-triggering SQL features unsupported by TxCheck (e.g., REPLACE), and TxCheck cannot determine that the DBMS performs wrongly for 3 txBugs.

6.5.2 End-to-End Experimental Comparison. We further perform an end-to-end comparison with the state-of-the-art transaction testing approaches, i.e., DT^2 [36], Troc [39] and TxCheck [45]. We do not directly compare WriteCheck with the transaction verification approaches, since they detect none of the critical txBugs in our conceptual comparison, indicating their ineffectiveness in detecting WSS violations.

The existing transaction testing approaches adopt cleverly-crafted test oracles for transaction test cases, which are usually complex and have constraints for target DBMSs. For example, DT^2 [36] utilizes differential testing, which requires the DBMSs under test to be compatible. Thus, it only supports MySQL-compatible DBMSs, and cannot test PostgreSQL, SQLite and CockroachDB. Therefore, we only run these approaches on their supported DBMSs, i.e., MySQL, MariaDB and TiDB, and do not extend them to support other DBMSs.

Table 4: End-to-End Experimental Comparison with Existing Transaction Testing Approaches

DBMS	WriteCheck			DT^2			Troc			TxCheck		
	Critical	Non-critical	False Positives	Critical (*)	Non-critical	False Positives	Critical (*)	Non-critical	False Positives	Critical (*)	Non-critical	False Positives
MySQL	0	0	1	0 (0)	0	24	1 (1)	1	2	0 (0)	0	1
MariaDB	1	0	1	1 (1)	1		0 (0)	1	1	0 (0)	0	2
TiDB	3	0	2	1 (1)	1		1 (1)	0	1	1 (1)	1	1
PostgreSQL	0	0	1	-	-	-	-	-	-	-	-	-
SQLite	0	0	0	-	-	-	-	-	-	-	-	-
CockroachDB	0	0	0	-	-	-	-	-	-	-	-	-
Total	4	0	5	2 (2)	2	24	2 (2)	2	4	1 (1)	1	4

* The numbers in parentheses show critical txBugs that can be detected by WriteCheck if the corresponding transaction test cases are provided to WriteCheck.

We run WriteCheck, DT^2, Troc and TxCheck on each of their supported DBMSs for 12 hours. Table 4 shows the experimental results. We do not compare the detection results for unsupported DBMSs of the corresponding approaches. We do not expect WriteCheck to detect more txBugs than existing approaches, since they can detect non-critical txBugs that WriteCheck cannot detect.

WriteCheck reports 4 critical txBugs, of which 1 and 3 are triggered in MariaDB and TiDB, respectively. WriteCheck reports 5 false positives, 2 of which are caused by semi-consistent read, and the other 3 are caused by lack of gap locks.

DT^2 reports 2 critical txBugs, 2 non-critical txBugs, and 24 false positives. The 24 false positives are caused by the incompatibility issues between DBMSs. Troc reports 2 critical txBugs, 2 non-critical txBugs, and 4 false positives. The 4 false positives are caused by inconsistent lock strategies in different DBMSs. TxCheck reports 1 critical txBug, 1 non-critical txBug, and 4 false positives. The 4 false positives are caused by the wrongly captured statement dependencies. For example, the motivating example discussed in TxCheck is confirmed by DBMS developers as an intentional and expected behavior, and not a txBug.

We further investigate whether WriteCheck can detect the critical txBugs reported by existing approaches by providing the bug-triggering transaction test cases to WriteCheck, and vice versa. Among the 4 critical txBugs detected by WriteCheck, 3 critical txBugs cannot be detected by existing approaches, because these approaches cannot support the involved bug-triggering SQL features. On the contrary, all the reported 2, 2 and 1 critical txBugs by DT^2, Troc and TxCheck can be detected by WriteCheck.

Benefited from the simplicity of WriteCheck, WriteCheck *can support more features in DBMSs, and thus detect more critical* txBug*s than the state-of-the-art approaches. All critical* txBug*s reported by existing approaches can also be detected by* WriteCheck.

7 DISCUSSION

False positives. We have proved that if the target DBMS satisfies Assumption 1 (Section 4), WriteCheck does not report false positives. But, if the target DBMS does not satisfy Assumption 1 (e.g., TiDB cannot identify predicate-based conflicts [19, 20]), WriteCheck can potentially report false positives. Identifying false positives involves transaction semantics in respective DBMSs, and we still cannot distinguish critical txBugs from false positives.

False negatives. Although WriteCheck is effective in detecting WSS violations in many DBMSs, it can still miss some critical txBugs due to the following four design choices. (1) WriteCheck may miss critical txBugs that involve unsupported SQL features (e.g., non-deterministic functions). (2) WriteCheck adopts random strategies to explore the huge input space of transaction testing, resulting in potential false negatives. (3) A buggy DBMS may generate an incorrect concurrent schedule for concurrent transactions, which may cause WriteCheck to infer an unexpected serial schedule. If this unexpected serial schedule coincidentally produces the same incorrect final database state as the concurrent schedule, a false negative will occur. (4) The transaction test cases in few (8.6% in Section 3.2) critical txBugs do not violate WSS. WriteCheck cannot detect these critical txBugs, too.

8 RELATED WORK

We introduce related works that we have not discussed yet.

DBMS testing. Many approaches focus on DBMS testing [16, 21, 22, 42, 44, 46, 47, 51–53, 58–64, 68, 69, 71, 72, 75–79]. SQLsmith [16] detects crash bugs by randomly generating SQL statements in DBMSs. Squirrel [78] tests DBMSs guided by code coverage. Rigger et al. propose several approaches, e.g., PQS [60], TLP [59] and NoREC [58], to detect logic bugs by constructing oracles for SELECT statements. Amoeba [53] compares execution time of two semantically equivalent queries to detect performance bugs. DQE [62] executes SELECT, UPDATE and DELETE statements with the same filter conditions, and any discrepancy indicates a logic bug. Differential testing has been effectively applied to test DBMSs [61, 72, 77]. These existing approaches cannot detect txBugs in DBMSs.

Transaction concurrency problems in DBMS-based applications. Transaction concurrency problems [29, 38, 54, 56, 67, 74] in DBMS-based applications, e.g., a shopping application built on MySQL [11], are caused by unintentional implementations of DBMS-based applications rather than DBMSs. Tang et al. [67] perform a comprehensive study on transaction concurrency problems caused by ad hoc transactions. Several approaches [29, 38, 54, 56, 74] have been proposed to detect transaction concurrency problems, e.g., Zellag et al. [74] build transaction dependency graphs to detect consistency anomalies. txBugs in DBMSs are orthogonal to transaction concurrency problems in DBMS-based applications.

9 CONCLUSION

We propose write-specific serializability, which is applicable on all isolation levels and concurrency control modes in DBMSs. Our study shows that write-specific serializability can be an effective test oracle to expose critical transaction bugs that can cause concurrent transactions to produce incorrect database states. We further present a simple and general transaction testing approach WriteCheck to detect write-specific serializability violations, and detect 11 new critical transaction bugs in production-level DBMSs.

ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China (62072444, 62302493), Major Program (JD) of Hubei Province (2023BAA018), Major Project of ISCAS (ISCAS-ZD-202302), Basic Research Project of ISCAS (ISCAS-JCZD-202403), Youth Innovation Promotion Association at Chinese Academy of Sciences (Y2022044), and Huawei.

REFERENCES

[1] 2025. The ANSI Isolation Levels. https://renenyffenegger.ch/notes/development/databases/SQL/transaction/isolation-level.

[2] 2025. CockroachDB. https://www.cockroachlabs.com.

[3] 2025. Current read in PostgreSQL. https://www.postgresql.org/docs/current/transaction-iso.html.

[4] 2025. Current read in TiDB. https://docs.pingcap.com/tidb/stable/pessimistic-transaction.

[5] 2025. DB-Engines. https://db-engines.com/en/ranking.

[6] 2025. Gap locks in MySQL. https://dev.mysql.com/doc/refman/8.0/en/innodb-locking.html#innodb-gap-locks.

[7] 2025. GitHub. https://github.com/.

[8] 2025. go-randgen. https://github.com/pingcap/go-randgen.

[9] 2025. MariaDB. https://mariadb.org.

[10] 2025. MySQL. https://www.mysql.com.

[11] 2025. MySQL Customer: Shopify. https://www.mysql.com/customers/view/?id=1303.

[12] 2025. PostgreSQL. https://www.postgresql.org.

[13] 2025. Semi-consistent read in MySQL. https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html.

[14] 2025. SQLancer. https://www.manuelrigger.at/dbms-bugs/.

[15] 2025. SQLite. https://www.sqlite.org/index.html.

[16] 2025. SQLsmith. https://github.com/anse1/sqlsmith.

[17] 2025. TiDB issues. https://github.com/pingcap/tidb/issues/.

[18] 2025. TiDB, PingCAP. https://pingcap.com.

[19] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[20] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of International Conference on Data Engineering (ICDE)*. 67–78.

[21] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 2060–2071.

[22] Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proc. ACM Manag. Data (SIGMOD)* 2, 3 (2024), 1–26.

[23] 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 (VLDB)* 7, 3 (2013), 181–192.

[24] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Transactions on Database Systems (TODS)* 41, 3 (2016), 15:1–15:45.

[25] 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)*. 1–10.

[26] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.

[27] 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.

[28] 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.

[29] 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.

[30] Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible Database Generators. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 1097–1107.

[31] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. 2006. Generating Queries with Cardinality Constraints for DBMS Testing. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 18, 12 (2006), 1721–1725.

[32] 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)*. 458–472.

[33] 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 (SIGFIDET)*. 249–264.

[34] Jack Clark, Alastair F. Donaldson, John Wickerson, and Manuel Rigger. 2024. Validating Database System Isolation Level Implementations with Version Certificate Recovery. In *Proceedings of the European Conference on Computer Systems (EuroSys 24)*. 754–768.

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

[36] 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 *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 35:1–35:12.

[37] Ziyu Cui, Wensheng Dou, Yu Gao, Dong Wang, Jiansen Song, Yingying Zheng, Tao Wang, Rui Yang, Kang Xu, Yixin Hu, Jun Wei, and Tao Huang. 2024. Understanding Transaction Bugs in Database Systems. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 163:1–163:13.

[38] Yuetang Deng, Phyllis Frankl, and Zhongqiang Chen. 2003. Testing Database Transaction Concurrency. In *Proceedings of IEEE International Conference on Automated Software Engineering (ASE)*. 184–193.

[39] Wensheng Dou, Ziyu Cui, Qianwang Dai, Jiansen Song, Dong Wang, Yu Gao, Wei Wang, Jun Wei, Lei Chen, Hanmo Wang, Hua Zhong, and Tao Huang. 2023. Detecting Isolation Bugs via Transaction Oracle Construction. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 1123–1135.

[40] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 243–252.

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

[42] Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, Wenpin Jiao, and Tao Xie. 2023. GDsmith: Detecting Bugs in Cypher Graph Database Engines. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 163–174.

[43] Daniel Jackson and Craig A. Damon. 1996. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering (TSE)* 22, 7 (1996), 484–495.

[44] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In *Proceedings of USENIX Security Symposium (USENIX Security)*. 4949–4965.

[45] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 397–417.

[46] 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.

[47] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. 2023. Testing Graph Database Engines via Query Partitioning. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 140–149.

[48] 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.

[49] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proceedings of the VLDB Endowment (VLDB)* 14, 3 (2020), 268–280.

[50] 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.

[51] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-Oriented DBMS Fuzzing. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*. 668–681.

[52] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *Proceedings of USENIX Security Symposium (USENIX Security)*. 4309–4326.

[53] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries. In *Proceedings of IEEE/ACM SIGSOFT International Conference on Software Engineering (ICSE)*. 225–236.

[54] Hao Luo, Mehedi Masud, and Hasan Ural. 2012. Detecting Offline Transaction Concurrency Problems. *Journal of Software (JSW)* 7, 8 (2012), 1855–1860.

[55] Dan R. K. Ports and Kevin Grittner. 2012. Serializable snapshot isolation in PostgreSQL. *Proceedings of the VLDB Endowment (VLDB)* 5, 12 (2012), 1850–1861.

[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. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 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 International Conference on Very Large Data Bases (VLDB)*. 618–622.

[62] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing Database Systems via Differential Query Execution. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 2072–2084.

[63] Jiansen Song, Wensheng Dou, Yu Gao, Ziyu Cui, Yingying Zheng, Dong Wang, Wei Wang, Jun Wei, and Tao Huang. 2024. Detecting Metadata-Related Logic Bugs in Database Systems via Raw Database Construction. *Proceedings of the VLDB Endowment (VLDB)* 17, 8 (2024), 1884–1897.

[64] Jiansen Song, Wensheng Dou, Yingying Zheng, Yu Gao, Ziyu Cui, Wei Wang, and Jun Wei. 2025. Detecting Schema-Related Logic Bugs in Relational DBMSs via Equivalent Database Construction. *Proceedings of the VLDB Endowment (VLDB)* 18, 7 (2025), 2281–2294.

[65] 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 (COMPSAC)*. 132–139.

[66] 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.

[67] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2022. Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly. In *Proceedings of International Conference on Management of Data (SIGMOD)*. 4–18.

[68] Lei Tang, Wensheng Dou, Yingying Zheng, Lijie Xu, Wei Wang, Jun Wei, and Tao Huang. 2025. Proving Cypher Query Equivalence. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*.

[69] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *Proceedings of International Conference on Software Engineering: Software Engineering in Practice (ICSE SEIP)*. 328–337.

[70] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc.

[71] Rui Yang, Ziyu Cui, Wensheng Dou, Yu Gao, Jiansen Song, Xudong Xie, and Jun Wei. 2025. Detecting Isolation Anomalies in Relational DBMSs. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[72] Rui Yang, Yingying Zheng, Lei Tang, Wensheng Dou, Wei Wang, and Jun Wei. 2023. Randomized Differential Testing of RDF Stores. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE Demo)*. 136–140.

[73] 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.

[74] Kamal Zellag and Bettina Kemme. 2014. Consistency Anomalies in Multi-tier Architectures: Automatic Detection and Prevention. *The VLDB Journal* 23, 1 (2014), 147–172.

[75] Yingying Zheng, Wensheng Dou, Lei Tang, Ziyu Cui, Yu Gao, Jiansen Song, Liang Xu, Jiaxin Zhu, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2024. Testing Gremlin-Based Graph Database Systems via Query Disassembling. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1695–1707.

[76] Yingying Zheng, Wensheng Dou, Lei Tang, Ziyu Cui, Jiansen Song, Ziyue Cheng, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2024. Differential Optimization Testing of Gremlin-Based Graph Database Systems. *Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 25–36.

[77] 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.

[78] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 58–71.

[79] Zeyang Zhuang, Penghui Li, Pingchuan Ma, Wei Meng, and Shuai Wang. 2024. Testing Graph Database Systems via Graph-Aware Metamorphic Relations. *Proceedings of the VLDB Endowment (VLDB)* 17, 4 (2024), 836–848.