Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction

Zu-Ming Jiang
ETH Zurich

Si Liu
ETH Zurich

Manuel Rigger
National University of Singapore

Zhendong Su
ETH Zurich

Abstract

Transactions are an important feature of database management systems (DBMSs), as they provide the ACID guarantees for a sequence of database operations. Consequently, approaches have been proposed to automatically find transactional bugs in DBMSs. However, they cannot handle complex operations and predicates common in real-world database queries, and thus miss bugs.

This paper introduces a general, effective technique for finding transactional bugs in DBMSs that supports complex SQL queries and predicates. At the conceptual level, we address the test-oracle problem by constructing semantically-equivalent test cases based on fine-grained statement-level dependencies in transactions. At the technical level, we introduce (1) *statement-dependency graphs* to describe dependencies among SQL statements in transactions, (2) *SQL-level instrumentation* to capture possible statement-level dependencies, and (3) *transactional oracle construction* to generate semantically-equivalent test cases using statement-dependency graphs. We also establish the correctness of our approach in generating semantically-equivalent test cases. We have realized our technique as a tool, TxCheck, and evaluated it on three widely-used and well-tested DBMSs, namely TiDB, MySQL, and MariaDB. In total, TxCheck found 56 unique bugs, 52 of which have been confirmed and 18 already fixed. We believe that TxCheck can help solidify DBMSs' support for transactions thanks to its generality and effectiveness.

1 Introduction

Database management systems (DBMSs) store and manage data and are crucial for many applications. A key feature of DBMSs is their support for transactions, where a sequence of SQL statements are executed as a single unit, and various properties (*i.e.*, atomicity, consistency, isolation, and durability) are guaranteed. For example, if some transactions are concurrently executed at the *Serializability* isolation level, uncommitted operations by other transactions will be invisible, and the transaction execution results must be equal to the results when these transactions are executed in a serial order. Benefiting from these properties, transactions have been applied in many critical applications. However, transaction implementations usually involve complex logic (*e.g.*, two-phase locking [7, 45] and multiversion concurrency control [8, 32]), and thus bugs are easily introduced. In this paper, we refer to the bugs in the transaction support of DBMSs as *transactional bugs*. Such bugs are critical because they can paralyze their client applications or, even worse, silently trigger incorrect behaviors in critical operations of client applications.

To improve the reliability and correctness of transaction processing in DBMSs, several approaches [4, 9, 11, 17, 44] have been proposed to test transaction support. These approaches use specific operation patterns to capture the violations of transactional rules. For example, ELLE [4] encodes transaction execution histories[1] by only appending to a conceptual list data structure. It builds transaction-dependency graphs based on the histories, and reports bugs if the graphs violate the desired isolation guarantees. Limited by specific test-case patterns, these approaches use only simple operations (*e.g.*, ELLE appends list with constant values), while many deep bugs may only be triggered by complex operations [21]. Moreover, existing approaches [4, 9, 17, 44] cannot handle predicates (*e.g.*, the condition expressions in **`WHERE`** clauses) in general. For example, ELLE cannot encode predicate operations in its list data structure. However, predicates are ubiquitous in real-world transactions as they rely on common features such as **`WHERE`** clauses or **`JOIN`**s. Their lack of predicate support makes existing approaches miss many real bugs. In addition, transactional bugs may be independent of isolation levels (*e.g.*, incorrect results returned by one transaction), while existing approaches focus on testing isolation levels, thus missing bugs.

Figure 1(a) shows a confirmed bug in MySQL at the *Repeatable Read* isolation level. The bug-triggering test case involves two tables and two interleaved transactions. The statement `T1.S3` is executed immediately after `T1.S2`. The-

[1] A history records transactional requests to and responses from a database.

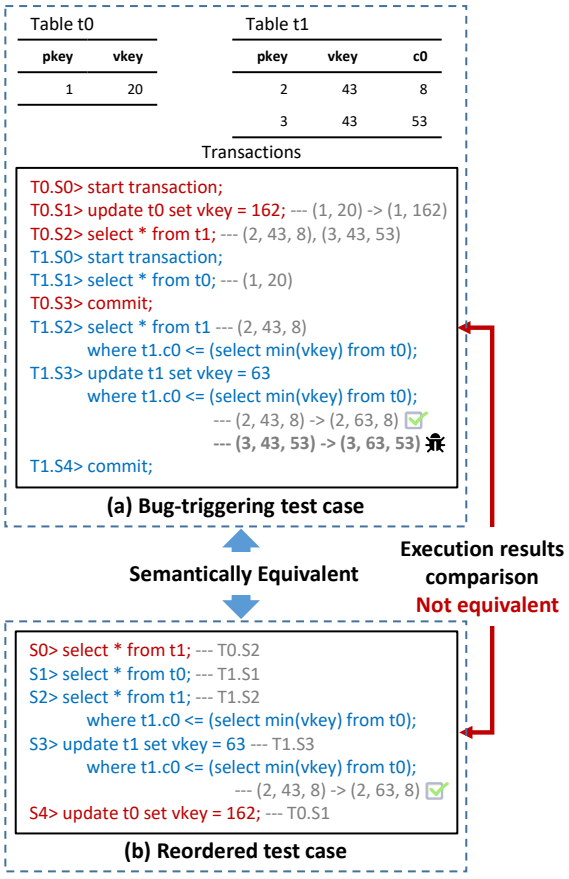

Figure 1: A MySQL bug found by TxCheck under the *Repeatable Read* isolation level.

oretically, `T1.S3` should fetch exactly those records subsequently updated by `T1.S2`, because they use the same predicate (*i.e.*, the same expression in their **`WHERE`** clause), and no other operations are executed between them. However, `T1.S2` fetches only the row (2, 43, 8), while `T1.S3` updates rows (2, 43, 8) and (2, 43, 53) due to its incorrect predicate matching. Existing approaches cannot find this bug for two reasons. First, the test case uses an aggregate function (*i.e.*, **`min`**`()`) and a subquery (*i.e.*, the **`SELECT`** in the **`UPDATE`** statement), which make the test case complex and not follow the test-case patterns of existing approaches (*e.g.*, ELLE can append only constant values instead of **`min`**`()` values). Second, the test case uses predicates, for which existing approaches lack support.

Figure 1(b) shows a test case generated by our approach, which is equivalent to the one shown in Figure 1(a). The three **`SELECT`** statements (*i.e.*, `S0`, `S1`, and `S2`) are moved before the two **`UPDATE`** statements (*i.e.*, `S3` and `S4`), because all these **`SELECT`** statements are oblivious to the effects of the **`UPDATE`** statements. As `T1.S3` is executed immediately after `T1.S2`, the database state visible for `T1.S2` and `T1.S3` should be consistent. Therefore, we keep the statements of `T1.S2` and `T1.S3` adjacent (*i.e.*, `S2` and `S3`). The reordered test case is executed *without using transactions*, and its execution results should be the same as the original one, because the reordering does not change any expected behavior of each statement. In this case, the bug in MySQL breaks the equivalence.

Our insight is to generate semantically-equivalent test cases that are not wrapped as transactions, but produce the same execution results, by properly reordering the statements. Then, we can validate whether their equivalence indeed holds. Any discrepancy indicates a bug in the target DBMS. While for the aforementioned test case, it is intuitive that reordering the statements will not affect the execution results of follow-up test cases, we must reason, in general, about the dependencies of statements. To this end, we first propose *statement-dependency graphs*, a novel concept to describe the dependencies among executed statements, which provide finer-grained dependency information than transaction-dependency graphs [1, 17, 44]. This facilitates finding more bugs (as will be discussed in Section 5.3). To extract statement dependencies, we propose *SQL-level instrumentation*. Specifically, we insert additional statements to collect the execution results of each target statement in transactions. Based on the collected results, we can track the operation effects—including the effects of predicate operations—of each statement, and thus infer all possible statement dependencies. To generate semantically-equivalent test cases, we propose *transactional oracle construction*. Specifically, we topologically sort the acyclic statement-dependency graphs, whose sorted statement sequences are proved to be semantically equivalent to the original one. To guarantee the acyclicity of graphs, we iteratively delete statements in cycles before sorting. We execute the sorted statement sequences without transactions and compare their results to those from the corresponding transaction executions. Any difference reveals a bug in the tested DBMS.

We realized this approach as a practical tool called TxCheck. We evaluate TxCheck on three popular and extensively tested DBMSs, namely TiDB [46], MySQL [30], and MariaDB [29]. In total, TxCheck found 56 unique bugs, including 23 in TiDB, 18 in MySQL, and 15 in MariaDB. Among them, 52 bugs have been confirmed, 18 fixed, and 8 assigned CVE IDs; 30 are triggered in transaction executions. These results collectively demonstrate that TxCheck can find latent transactional bugs in mature production DBMSs.

Overall, we make the following contributions:

- At the conceptual level, we address the test-oracle problem of DBMS transaction testing by constructing semantically-equivalent test cases.
- At the technical level, we propose (1) statement-dependency graphs, which describe the dependencies among statements in executed transactions, (2) SQL-level instrumentation, which can capture all possible statement dependencies including the predicate-related dependencies, and (3) transactional oracle construction, which refines test cases and generates semantically-equivalent test cases for validation. We formally prove the correctness of our approach.

- At the practical level, we implement our approach into a tool, TxCheck, and evaluate it on three widely-used DBMSs (*i.e.*, TiDB, MySQL, and MariaDB). In total, TxCheck finds 56 unique bugs, most of which cannot be identified by existing approaches. TxCheck is open-sourced at https://github.com/JZuming/TxCheck.

2 Background

Transactions in DBMSs. A database transaction refers to a series of operations, for which DBMSs must guarantee atomicity, consistency, isolation, and durability (*i.e.*, *ACID*) [49]. This paper focuses on relational database management systems (RDBMSs), where a transaction typically consists of a group of SQL (*i.e.*, Structured Query Language) statements. Each statement performs read operations (*e.g.*, **`SELECT`** statements) or write operations (*e.g.*, **`INSERT`**, **`DELETE`**, and **`UPDATE`** statements). SQL statements commonly involve predicates (*e.g.*, **`WHERE`** clauses) for choosing desired rows that satisfy the requirements.

Dependency Graphs. Adya *et al.* [1, 2] propose transaction dependencies, which can be classified into two categories, namely *item dependencies* and *predicate dependencies*. Item dependencies describe the relationship among transactions on specific items (*i.e.*, rows in tables) they read from or write to. For example, transaction T_j *item-read-depends* on T_i if T_j reads an item version x_i that is written by T_i. A predicate dependency describes the relationship between two transactions constructed from the associated predicate operations. For example, transaction T_j *directly predicate-read-depends* on T_i if an item version x_i that is written by T_i is used for predicate matching of T_j.

Based on these dependencies, Adya *et al.* propose a transaction-dependency graph, called *Direct Serialization Graph* (DSG). DSGs can be used to formalize the expected behaviors of DBMSs under different isolation levels. For example, *Serializability* (PL-3) proscribes any directed cycle in a DSG, while *Repeatable Read* (PL-2.99) any directed cycles that dismiss certain predicate dependencies. Bailis *et al.* [5, 6] further extend DSGs to define several other isolation levels.

Existing Approaches. Both transaction-focused testing and verification approaches [4, 9, 11, 17, 44] have utilized dependency graphs. They typically use specific operation patterns to capture transaction dependencies. For example, to reduce the search space of possible transaction orders, COBRA [44] exploits the read-modify-write (RMW) patterns where a transaction reads from a key before writing to it. By restricting its writes to list-specific operations like "append", ELLE [4] can naturally infer the transaction order from a list of values read.

While existing approaches [4, 9, 17, 44] such as ELLE have been successful in detecting a wide range of important bugs, they are limited in finding *deep* transactional bugs for two main reasons. First, these approaches can use only simple operations (*e.g.*, writing key-value pairs) following their restricted operation patterns. However, many deep DBMS bugs can only be triggered by complex operations [21]. Second, existing approaches lack support for predicate operations in general. In contrast to read/write operations whose effects are explicitly reflected in the final execution results, the effects of predicate operations are implicit and difficult to track, because they are typically reflected in the intermediate processes (*e.g.*, choosing *a set of items* that satisfy the predicate conditions for subsequent read/write operations). However, predicates are commonly used in real-world transactions and involved in sophisticated features of DBMSs like predicate optimization [23]. Existing approaches do not consider predicate operations in their test cases, thus missing many bugs with respect to the transaction support of DBMSs. In addition, existing approaches focus on testing isolation guarantees while many transactional bugs are not necessarily related to database isolation.

3 Approach

In this section, we present a novel approach for addressing the challenges of testing transaction support in DBMSs, as illustrated in Figure 2. Our core idea is to extract the dependencies among statements in transactions and construct semantically-equivalent test cases according to the extracted dependency information. The constructed test cases are used as oracles to validate the original transaction executions by checking whether all statements in the test cases produce the same results. To realize this idea, we first define statement dependencies and propose a new concept, *statement-dependency graphs*, which describes the dependencies among statements in transactions. To derive statement-dependency graphs, we capture dependency information from specific transaction executions. Then, we generate semantically-equivalent test cases based on the captured information.

Dependency Capturing. To capture statement dependencies, we propose *SQL-level instrumentation*, which inserts SQL statements to collect execution information of each original statement in transactions. Using SQL to achieve this makes this approach a *black-box technique* that is applicable even for DBMS where testers lack access to the source code. Specifically, we instrument in two steps, item-tracking instrumentation and version-set-tracking instrumentation, which capture item dependencies and predicate dependencies, respectively. Statement-dependency graphs are built based on the outputs from the inserted and the original statements.

Oracle Construction. We propose *transactional oracle construction* to construct semantically-equivalent test cases. We first iteratively remove statements involved in cycles to make the statement-dependency graph acyclic, which is the precondition of the construction. Then, we perform topological sort-

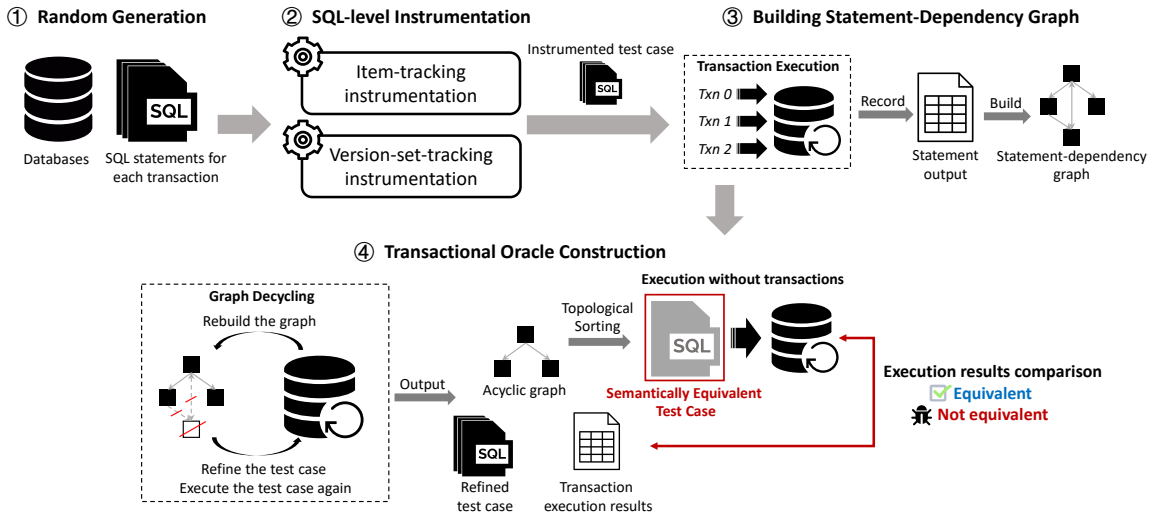

Figure 2: Approach overview.

ing on the acyclic graph to construct semantically-equivalent test cases. We execute these test cases without transactions and compare their execution results to those from the corresponding transaction executions. We prove that, for any correct DBMS, these results should be identical, thus any difference indicates an actual DBMS bug.

3.1 Statement-Dependency Graph

To construct semantically-equivalent test cases by reordering statements, we need to identify the dependencies between statements. We define seven kinds of statement dependencies, referring to transaction dependencies defined by Adya *et al.* [1, 2]. Each kind of statement dependency is shown in Figure 3. Specifically, we define three kinds of statement dependencies, shown in Figure 3(a)–(c), to model the relationship of two statements that read or write the same items. Three other kinds of dependencies, shown in Figure 3(d)–(f), are used to represent the dependencies established by predicate operations. The last one, direct stmt-value-write dependency shown in Figure 3(g), is used to model the event that the new value of an item installed by a statement is determined by the value of another item that is installed by another statement. The formal definitions (Definition 2–8) for these statement dependencies are given in Appendix A. Note that we do not count statement orders as dependencies, because when two statements access different data, the statements' execution order will not affect their results.

In contrast to the dependencies defined by Adya *et al.* [1,2], which describe the relationship between transactions, our definitions model the relationship between statements to provide finer-grained dependency information. Statement dependencies enable us to analyze the effects of each statement, which is needed for generating semantically-equivalent test cases accordingly, while transaction dependencies lack sufficient information related to statements. Next, we further propose statement-dependency graphs, *SDG*, to model the executions of test cases at the statement level.

Definition 1 (Statement-Dependency Graph) We define the statement-dependency graph constructed based on a statement execution history H, denoted as *SDG(H)*, as follows. *SDG(H)* is a directed graph, whose nodes represent the statements in committed transactions, and whose (directed) edges represent the dependencies between these committed statements. In particular, if statement S_j depends on statement S_i, there is a direct edge from S_i to S_j.

Note that statement-dependency graphs consider only statements in committed transactions. The statements in aborted transactions are dismissed because these statements conceptually do not affect the manipulated databases and other committed transactions. Figure 4 shows the statement-dependency graph for the test case in Figure 1(a). Statement-dependency graphs contain all dependency information related to statements and reflect the execution results of test cases at the statement level, which is the basis for generating semantically-equivalent test cases.

3.2 SQL-Level Instrumentation

To build statement-dependency graphs, we extract statement dependencies from transaction executions. In contrast to existing work, we aim to support test cases whose statements use predicates, without involving too many restrictions on test-case patterns. This section presents *SQL-level instrumentation*, a novel technique for extracting statement dependencies from transaction executions involving predicates.

The basic idea of SQL-level instrumentation is to insert statements to output the handled items before and after the operations performed by target statements. To realize this

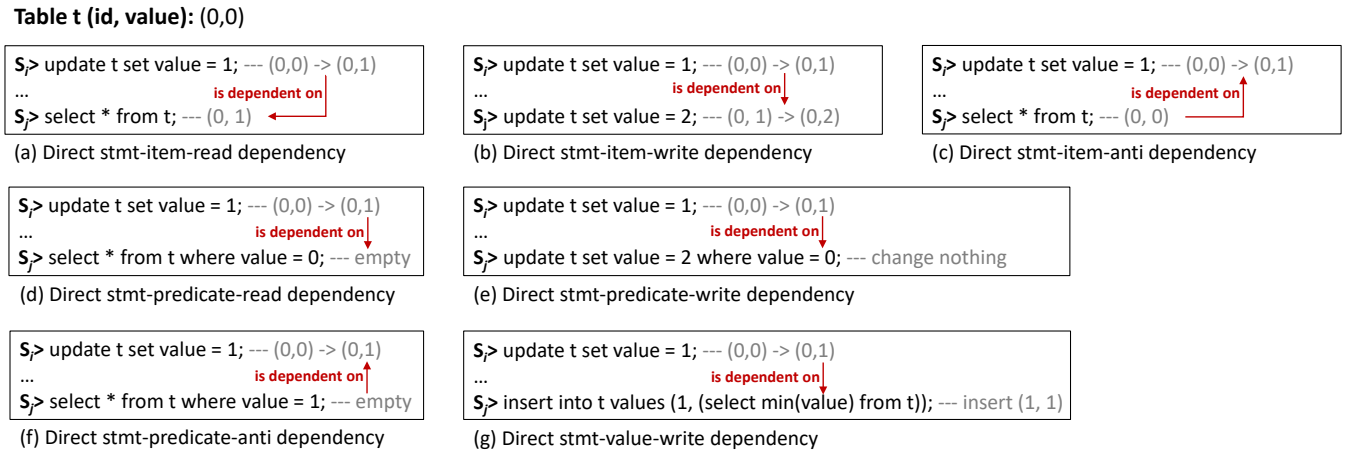

Figure 3: Examples for each kind of statement dependency.

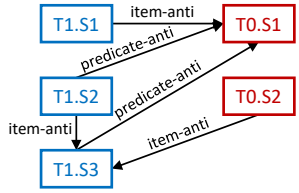

Figure 4: Statement-dependency graph for Figure 1(a).

idea, SQL-level instrumentation requires tables in manipulated databases to contain at least two columns. We name these two columns as *PrimaryKey* column and *VersionKey* column, respectively. The column *PrimaryKey* is used to identify different items, and should not change after the items are inserted. The column *VersionKey* is used to identify different versions of each item, and should be assigned a new value different from any earlier values when the item is updated by statements. Besides these two, tables may have additional columns whose properties are unrestricted.

The statements in test cases should also follow these restrictions. Specifically, the statement performing write operations (*e.g.*, **`UPDATE`** and **`INSERT`**) should change the *VersionKey* of the handled items to a new value, and each item updated by the same statement has the same *VersionKey*. The statement performing read operations (*e.g.*, **`SELECT`**) should at least output the *PrimaryKey* and *VersionKey* of the items. Except for the *PrimaryKey* and *VersionKey* of items, we eschew imposing any additional restrictions for the generated statements.

SQL-level instrumentation operates in two phases: (1) item-tracking instrumentation, and (2) version-set-tracking instrumentation. Figure 5 shows how each phase instruments the test case in Figure 1. In item-tracking instrumentation, we insert a *Before-Write Read* (*BWR*) statement before each statement performing write operations (*e.g.*, see `T0.S1.BWR` and `T1.S3.BWR` in Figure 5). *BWR* statements are designed to output the items that will be written and thus use the same predicates as the target statements. *BWR* statements can work only under isolation levels that satisfy Assumption 3, which

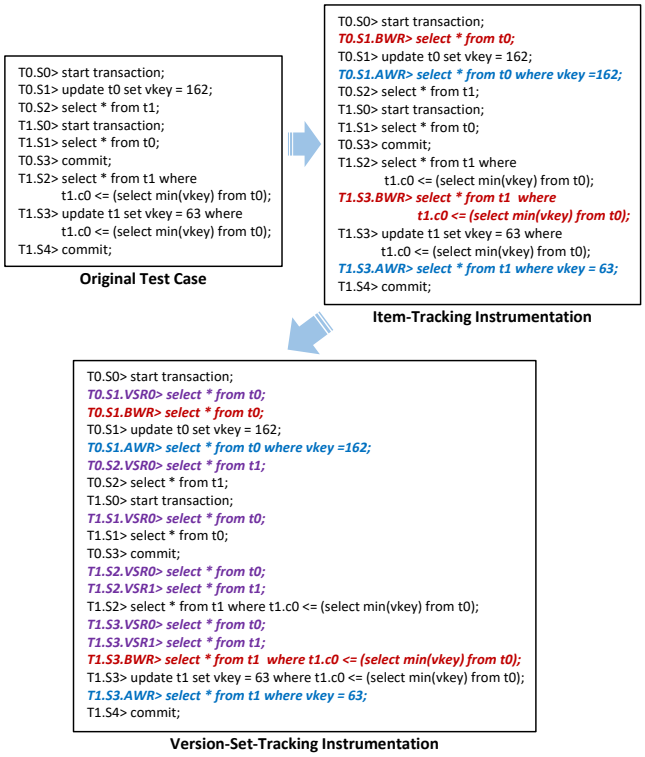

Figure 5: SQL-level instrumentation for Figure 1(a).

is discussed subsequently. We insert an *After-Write Read* (*AWR*) statement after each statement performing write operations (*e.g.*, see `T0.S1.AWR` and `T1.S3.AWR` in Figure 5). *AWR* statements are used to output the new values of the items processed by target statements. To do so, *AWR* statements select items whose *VersionKey*s are equal to the assigned values in the target statements. In version-set-tracking instrumentation, we insert some *Version-Set Read* (*VSR*) statements before each statement (*e.g.*, see `T0.S1.VSR0` and `T0.S2.VSR0` in Figure 5). To output the item versions referenced by target statements, *VSR* statements output all items in the tables ref-

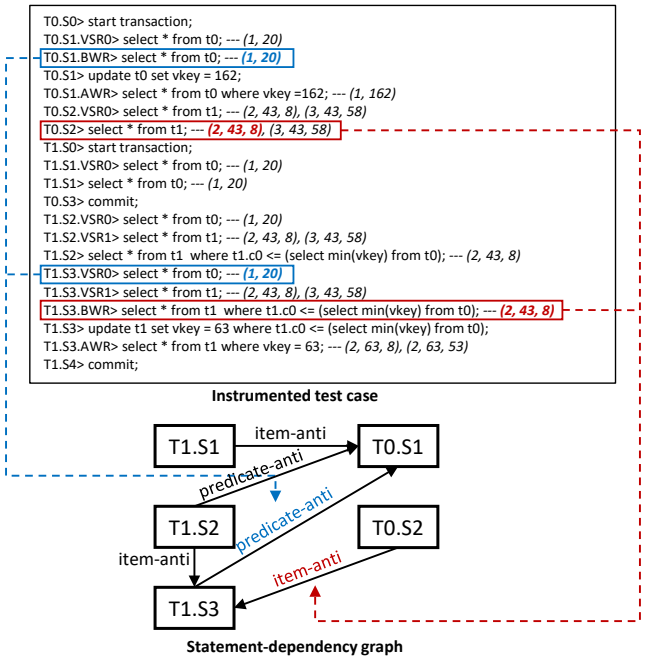

Figure 6: Inferring statement-dependency graphs for Figure 1.

erenced by the target statements.

Using the inserted *BWR*, *AWR*, and *VSR* statements, we can collect the execution information of each statement in transactions, and thus can infer possible statement dependencies. Figure 6 shows how the inserted statements are used to infer statement dependencies for the test case in Figure 1. To check whether statements `T1.S3` and `T0.S2` have dependencies between each other, we analyze the outputs of the corresponding statements and their inserted statements. The output of `T0.S2` and the output of *BWR* of `T1.S3` have an overlapping part (*i.e.*, item (2, 43, 8)), which means `T0.S2` reads an item that has not been updated by `T1.S3`. Therefore, `T0.S2` is (stmt-item-anti) depended on `T1.S3`. The outputs of *BWR* of `T0.S1` and *VSR*s of `T1.S3` also have an overlapping part (*i.e.*, item (1, 20)), which means that `T0.S1` will update an item that has been referenced by the predicates of `T1.S3`. Therefore, `T1.S3` is (stmt-predicate-anti) depended on `T0.S1`. Other dependencies can be inferred similarly.

We prove that each statement dependency proposed in Section 3.1 can be inferred based on the outputs of statements under certain assumptions. The detailed proof (Lemma 1–7) is presented in Appendix B. The assumptions are shown below:

Assumption 1 No synchronization issues happen during the execution of transactions.

Assumption 2 Statements can use item versions only in the tables that they have referenced.

Assumption 3 For any two transactions, T_i and T_j, it is prohibited that T_i item-anti-depends on T_j for the item x while T_j item-write-depends on T_i for the same item x.

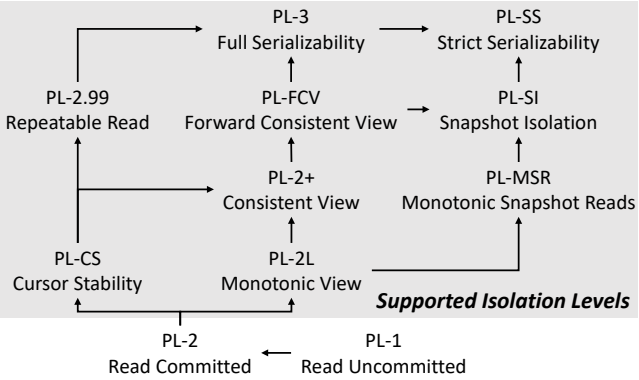

Figure 7: Isolation-level hierarchy defined by Adya *et al.* [1,2] and isolation levels supported by our approach.

These assumptions generally hold when we test DBMSs deployed locally (*i.e.*, without synchronization issues) with a proper isolation guarantee, which can be satisfied by the isolation levels equal to or stronger than Cursor Stability (PL-CS) or Monotonic View (PL-2L), according to Adya *et al.*'s definitions [1,2]. Figure 7 shows the supported isolation levels. Specifically, Assumptions 1 and 2 are independent of isolation levels. In a transaction dependency graph, Cursor Stability prohibits cycles with an anti-dependency and one or more write-dependency edges such that all edges are related to one specific object. Assumption 3 prohibits cycles with exactly one anti-dependency edge and one write-dependency edge such that both edges are related to one specific object. Therefore, Cursor Stability satisfies Assumption 3. Monotonic View disallows cycles containing exactly one anti-dependency edge from one transaction to another transaction. It satisfies Assumption 3, because the phenomenon prohibited by Assumption 3 contains cycles with exactly one anti-dependency edge between transactions.

SQL-level instrumentation can accurately capture item dependencies without any false positives or negatives, but it may capture spurious predicate or value dependencies, according to Lemma 1–7. Such inaccuracies may be introduced by *VSR* statements; *VSR* statements output all items in the tables referenced by target statements. However, target statements may use only a part of the items in the tables to perform their predicate matching and value capturing, depending on the specific implementations of the DBMS. Therefore, *VSR* statements may output items that are not referenced by target statements. If these incorrectly outputted items match the outputs of other statements, spurious dependencies are captured. Therefore, the statement-dependency graph built by SQL-level instrumentation is a super graph of the actual statement-dependency graph. This issue does not affect the correctness of our testing approach (see Section 3.3 for the detailed discussion).

We further discuss the time complexity of using SQL-level instrumentation to infer dependencies. Suppose the database contains n items, and the test case contains m statements. In

the worst case, each *BWR* or *AWR* statement can output n items, and the *VSR* statements of each target statement can output, in total, n items. If the target statement is a read statement, it can output at most n items. When both two target statements are write statements, we need to check $6n$ items (*i.e.*, outputs of *BWR*, *AWR*, and *VSR* statements of each target statement), while we need only check $5n$ items when one of the target statements is a read statement (*i.e.*, outputs of *BWR*, *AWR*, and *VSR* statements of the write statement and outputs of the read statement and its *VSR* statements). Therefore, in the worst case, we may check whether two target statements have dependencies in $O(6n)$ time using hash tables. To confirm the dependencies among m statements, we should check $m(m-1)$ dependencies, and thus the worst-case time complexity of the entire process is $O(6n{\cdot}m(m-1))$, *i.e.*, $O(m^2n)$.

Existing work proves that checking histories in isolation levels is a polynomial-time (*e.g.*, *Read Committed*) or even NP-complete problem (*e.g.*, *Serializability* and *Snapshot Isolation*) [9, 33]. ELLE [4] can recover histories in $O(m{\cdot}p)$, where m is the number of operations, and p is the number of concurrent processes. However, ELLE restricts their test cases whose write operations can only append and cannot handle histories involving predicates. In contrast, SQL-level instrumentation can recover histories involving predicates and only requires test cases to maintain *PrimaryKey* and *VersionKey* for each item, within $O(m^2n)$ time.

3.3 Transactional Oracle Construction

The statement-dependency graphs enable us to construct semantically-equivalent test cases. Our intuition is that if there is a reordered statement sequence whose statements follow the same dependency order in the statement dependency graph, the reordered statement sequence should produce the same results as the original one. To effectively test DBMSs' transaction support, we execute the reordered statements without transactions, which provides an oracle for validating the original test cases by checking whether each statement in the test cases produces the same results. To formalize our intuition, we first introduce a theorem based on statement-dependency graphs. The theorem is given below, and the details of its proof can be found in Appendix C.

Theorem 1 *SDG(H)* is the statement-dependency graph built according to execution history H, *S(SDG(H))* is the statement sequence generated by performing topological sorting on *SDG(H)*, and *History(S(SDG(H))* is the history of the sorted statement sequence executed without transactions. If *SDG(H)* is acyclic, *History(S(SDG(H)))* and H give the same results for each statement in *S(SDG(H))*.

Theorem 1 suggests a high-level method for constructing semantically-equivalent test cases. Theorem 1 is not constrained to any specific isolation level and thus can apply at various isolation levels. Moreover, Theorem 1 can tolerate the

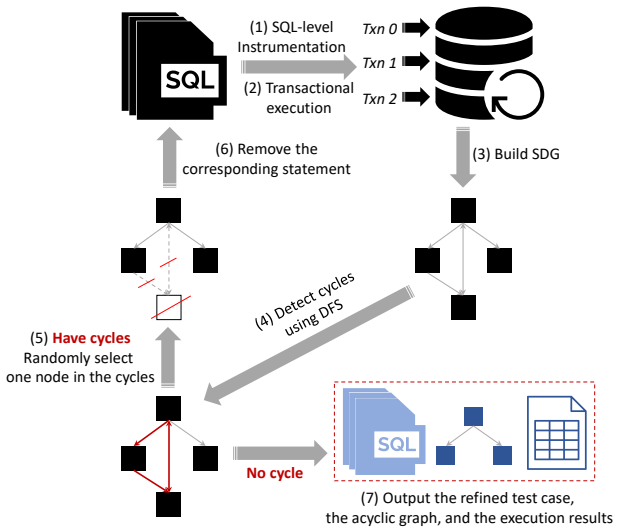

Figure 8: The process of refining test cases, whose SDG eventually becomes acyclic.

spurious predicate dependencies that stem from SQL-level instrumentation. Suppose G is the actual statement dependency graph and G' is G with additional spurious predicate dependencies, i.e., G is a subgraph of G'. The semantically-equivalent test case constructed from G' will follow all the dependencies in G', thus following all the dependencies in G. Hence, the constructed test case is also one of the topological sorting results of G. That is, the test case constructed from G' can also serve as an oracle.

Note that Theorem 1 has a precondition, *i.e.*, the *SDG(H)* should be acyclic. To satisfy this precondition, we perform *graph decycling* to eliminate all cycles in the graph. Then, we perform *oracle checking* to generate semantically-equivalent test cases by topologically sorting acyclic graphs. We execute these equivalent test cases without transactions and use their results to validate transaction executions.

Graph Decycling. The overview of graph decycling is shown in Figure 8. The idea is to break cycles in the graph by removing those statements involved in the cycles. Given a test case, we first instrument it (Step (1) in Figure 8) and then execute the instrumented test case using transactions (Step (2)). We can infer the SDG using the collected information from instrumented statements (Step (3)). Then, we check whether there is a cycle in the constructed SDG using depth-first search [39] (Step (4)). If there is at least one cycle in the graph, we randomly select one node in the cycles (Step (5)) and remove the corresponding statement in the test case (Step (6)). We re-execute the refined test case on the DBMS and start a new round of test-case refinement. Note that the re-execution is necessary, because a refined test case may result in the construction of a significantly different SDG, which might also contain new cycles. At the beginning of each round, the tested DBMS is reset to its initial state. When no cycle is detected in the SDG built from the refined test case, we output the refined

Algorithm 1: Oracle Checking

```
   input : test_case, graph, t_results
1  Function OracleChecking(test_case, graph, t_results, DBMS):
2      oracle_test_case ← OracleGen(test_case, graph);
3      DBMS ← INITIAL_STATE;
4      o_results ← ExecuteWithoutTxn(oracle_test_case, DBMS);
5      for each stmt in oracle_test_case do
6          t_stmt_results ← GetStmtResults(stmt, t_results);
7          o_stmt_results ← GetStmtResults(stmt, o_results);
8          if t_stmt_results ≠ o_stmt_results then
9              ReportBug();
10             return FALSE;
11     t_db ← GetDBContent(t_results);
12     o_db ← GetDBContent(o_results);
13     if t_db ≠ o_db then
14         ReportBug();
15         return FALSE;
16     return TRUE;
17 Function OracleGen(test_case, graph):
18     oracle_test_case ← [];
19     tmp_graph ← graph;
20     while HasNode(tmp_graph) = TRUE do
21         nodes ← GetZeroIndegreeNodes(tmp_graph);
22         node ← RandomlySelect(nodes);
23         stmt ← GetStmtFromNode(node, test_case, graph);
24         PushToList(oracle_test_case, stmt);
25         RemoveNode(tmp_graph, node);
26     return oracle_test_case;
```

test case, its corresponding acyclic SDG, and its transaction execution results (Step (7)). The graph decycling always converges, because it cannot indefinitely remove statements from the test case where the number of statements is finite.

Oracle Checking. Algorithm 1 shows the workflow of oracle checking. The inputs to this workflow are the refined *test case*, the acyclic *graph*, and the transaction execution results *t_results* from graph decycling. We first generate a semantically-equivalent test case, that is, *oracle_test_case*, using topological sorting (Line 2). Specifically, we initialize *oracle_test_case* as an empty sequence and *temp_graph* as *graph* (Line 18–19). In each round, we randomly select one node whose in-degree—the number of edges directed into the node—is zero and append the statement corresponding to this node to the end of *oracle_test_case* (Line 21–24). Then, we remove the node from *temp_graph* and delete the edges incident to this node (Line 25). The loop terminates if all nodes in *temp_graph* are removed (Line 20), and the process returns the constructed *oracle_test_case* (Line 26). After *oracle_test_case* is available, we initialize the target DBMS and execute the statements in *oracle_test_case* in order without transactions (Line 3–4). According to Theorem 1, for any correct DBMS implementation, *test_case* and *oracle_test_case* should produce the same result. We first compare each statement output (Line 5–10). If their outputs differ, we have found a bug. If these comparisons succeed, we further check whether the final database contents are the same (Line 11–15). If they are different, a bug is also found.

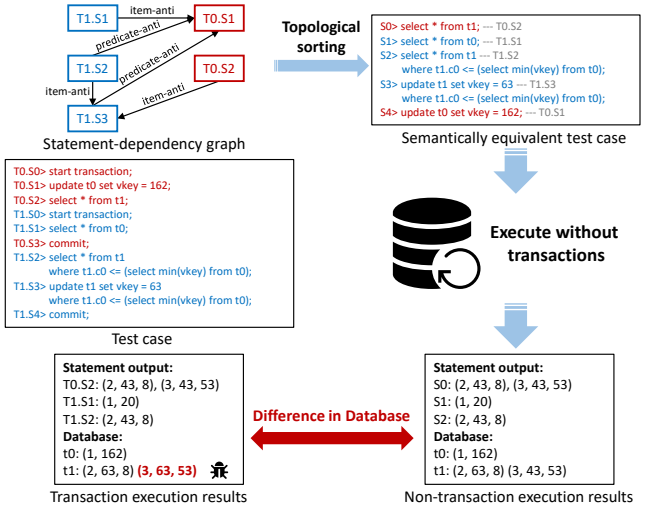

Figure 9: Oracle checking for the bug in Figure 1.

Figure 9 shows how we perform oracle checking on the test case in Figure 1(a). We first perform topological sorting on the acyclic statement-dependency graph. In the first round, `T1.S1`, `T1.S2`, and `T0.S2` have zero in-degree, so we randomly pick one of them, for example, `T0.S2`. Then, `T1.S1` and `T1.S2` are picked. After `T0.S2` and `T1.S2` are removed from the graph, the in-degree of `T1.S3` becomes zero, and `T1.S3` is picked. Finally, `T0.S1` is chosen as it is the only node in the graph. Therefore, the sorted statement sequence is [`T0.S2`, `T1.S1`, `T1.S2`, `T1.S3`, `T0.S1`]. Then, we execute the statement sequence without transactions and record the statement outputs and final database contents. These results are compared to the results produced by the original test case. We first check their statement outputs, which turn out to be the same. Then, we check their final database contents. Because their database contents are different on the value of one of the rows in table `t1`, we have found a bug.

4 Implementation

Based on our approach, we realized a tool, TxCheck, on top of SQLsmith [43]. The overall codebase consists of 14k lines of C++ code, where we implemented our approach with 3.5k lines (not including the code for supporting DBMSs).

Figure 10 shows the architecture of TxCheck. To test a DBMS, TxCheck first randomly generates a test case, which will be instrumented by SQL-level instrumentation. The instrumented test case is then refined by graph decycling to eliminate cycles in its statement-dependency graph, and by blocking scheduling to make sure that the instrumented statements will not be reordered by the blocking mechanism of the tested DBMS. TxCheck uses the refined test case and its transaction execution results to construct an oracle, which is a test case that is not wrapped as transactions (*e.g.*, the test case shown in Figure 1(b)), but should produce the same results

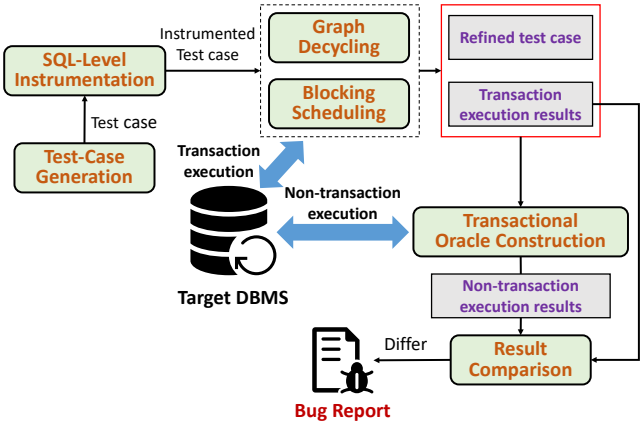

Figure 10: Architecture of TxCheck.

as the refined test case, according to Theorem 1. TxCheck then checks whether their results are indeed the same. If their results differ, TxCheck reports a bug. The following describes the implementation of TxCheck in detail.

Test-case Generation. TxCheck randomly generates a test case consisting of multiple transactions, and randomly determines the order in which the statements in these transactions are executed. For example, in Figure 1(a), `T0` and `T1` are the generated transactions, and [`T0.S0`, `T0.S1`, `T0.S2`, `T1.S0`, `T1.S1`, `T0.S3`, `T1.S2`, `T1.S3`, `T1.S4`] is the execution order for the statements in these transactions. The generated statements follow the constraints described in Section 3.2. As we do not restrict the statement format, TxCheck can apply other approaches to implementing statement generation. In this paper, we use SQLsmith [43] as the statement generator.

Transaction Execution. For each transaction in a test case, TxCheck sets up a client session that is responsible for issuing the statements of a transaction to which it is assigned. To avoid introducing non-determinism from concurrent executions, TxCheck sends statements to the DBMS following the order determined in the test-case generation. The order might be updated by block scheduling. After sending a statement, TxCheck sends the next statement only after the DBMS indicates that its execution is completed or blocked. While some concurrency bugs may be missed, sequential execution makes it significantly easier to reproduce bugs, which is generally appreciated by developers.

Non-transaction Execution. For each test case that is not wrapped as transactions (*e.g.*, the test case in Figure 1(b)), TxCheck sets up only one client session for sequentially issuing the statements in the test case.

Blocking Scheduling. When statements in different transactions try to access the same data, a DBMS may block some of these statements to schedule transaction execution, which can disrupt the inserted statements of SQL-level instrumentation. Figure 11(a) shows an example in MySQL using *Repeatable Read* isolation level. `T1.S1.BWR` is the *BWR* statement of

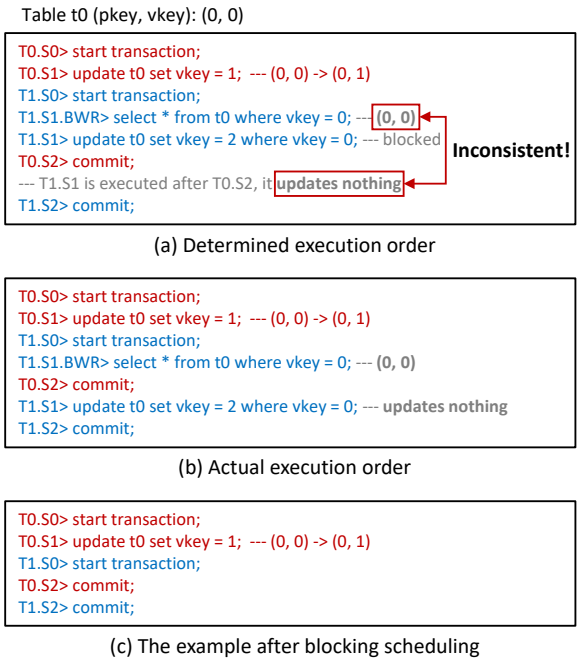

Figure 11: Example of MySQL blocking mechanism (in *Repeatable Read*) and blocking scheduling.

`T1.S1`. The DBMS executes `T1.S1.BWR` and outputs 1 row, and then tries to execute `T1.S1`. `T1.S1` is blocked, because it tries to update the items that are being updated by `T0.S1`. The DBMS continues to execute `T0.S2`. After `T0.S2` is executed, transaction `T0` is finished, and then `T1.S1` is executed automatically by the DBMS. `T1.S1` updates nothing because there is no row whose `vkey` is 0. By design, `T1.S1.BWR` should output the items that will be updated by `T1.S1`. However, their results are inconsistent because `T1.S1.BWR` is executed before `T0` commits but `T1.S1` is executed after `T0` commits. Figure 11(b) shows the actual execution order of the example. `T1.S1.BWR` and `T1.S1` are separated by the `COMMIT` of `T0`.

To address the issues caused by the blocking mechanism of DBMSs, TxCheck adapts blocking scheduling, which makes sure that the inserted statements and the target statements will not be separated. TxCheck first executes statements according to the determined execution order. It records the actual statement execution order, which may be different from the determined order because some statements may be blocked. It checks whether there are situations where the inserted statements and their corresponding target statements are executed apart. It deletes the inserted statements and target statements in such situations. For example in Figure 11, `T1.S1.BWR` and `T1.S1` are deleted. The refined test case is executed again following the recorded execution order in the last round. In the new execution, if all the target statements and corresponding inserted statements are executed adjacently, the blocking scheduling ends. Otherwise, it deletes statements according to the newly recorded real execution order and executes the re-

fined test case again. This process always converges, because it cannot delete statements indefinitely. Figure 11(c) shows the example refined by blocking scheduling.

DBMS Support. TxCheck can be easily adapted to test specific DBMSs. On average, we use 650 lines of code to support one DBMS. Each tested DBMS should provide interfaces to set up the DBMS, connect to the DBMS, shut down the DBMS, send statements in transaction sessions, and obtain execution results. In addition, if a DBMS can block statements in transaction execution, TxCheck needs to be provided with an interface for determining whether a statement is blocked. Setting a timeout for statements is an alternative way to check for blocking situations. However, it is inaccurate, because the DBMS may just be executing a long-running statement.

Isolation Bug Detection. TxCheck can also find isolation bugs, because statement dependencies can be easily converted to transaction dependencies according to their definitions. For example, if a statement S_i in transaction T_i depends on statement S_j in transaction T_j, T_i depends on T_j. Therefore, TxCheck can convert statement-dependency graphs to transaction-dependency graphs. Then, TxCheck detects bugs that violate their isolation levels according to graph restrictions [1, 2, 5, 6]. However, because TxCheck may infer spurious predicate dependencies (Section 3.2), which introduce false alarms of isolation bugs, we only consider item dependency, which is accurate, in isolation bug detection.

Memory Bug Detection. As TxCheck involves both transaction and non-transaction executions, memory bugs triggered with or without transactions can be detected by TxCheck. We use ASan [40] as its memory bug checker.

5 Evaluation

We have evaluated TxCheck on three DBMSs, namely TiDB [46], MySQL [30], and MariaDB [29]. These DBMSs are widely used by industry and extensively tested by DBMS fuzzers [21, 22, 25, 35–37, 43, 53]. According to DB-Engines Ranking [13], MySQL is the second most popular relational DBMS, MariaDB the 8th, and TiDB the 49th. The GitHub repositories of TiDB, MySQL, and MariaDB have been starred more than 32K, 8K, and 4K times, respectively, demonstrating their popularity and maturity. We perform our evaluation on Ubuntu 20.04 with a 64-core AMD Epyc 7742 CPU at 2.25G Hz and 256GB RAM.

We evaluated TxCheck on the latest available releases of the targeted DBMSs. Specifically, for TiDB, we tested versions 5.4.0, 6.1.0, and 6.3.0, for MySQL, versions 8.0.28 and 8.0.30, and for MariaDB, versions 10.8.3 and 10.10.1. We tested MySQL and MariaDB under *Read Committed*, *Repeatable Read*, and *Serializability*, respectively. We did not test *Read Uncommitted* because it does not satisfy Assumption 3 (see Figure 7). TxCheck can be used to test *Read Committed* in MySQL (as also in MariaDB) because it supports consistent nonlocking reads [31] (e.g., read operations of SELECT), thus satisfying Assumption 3. We tested TiDB with its optimistic transaction mode and *Snapshot Isolation*, which is the only isolation level compatible with this mode [48]. We did not test the pessimistic transaction mode of TiDB [47], which does not satisfy Assumption 3.

Table 1: Numbers of bugs found by TxCheck and their status

DBMS	Found	Confirmed	Known	Fixed
TiDB	23	19	1	9
MySQL	18	18	1	3
MariaDB	15	15	4	1
Total	56	52	6	13

We used TxCheck to continuously test the targeted DBMSs for three months; we stopped and restarted TxCheck only when we improved TxCheck with new SQL features. In general, TxCheck was able to find new bugs within several days after we implemented new features; however, certain bugs took more time to trigger (*e.g.*, one or two weeks).

5.1 Bug Detection

As shown in Table 1, TxCheck found 56 unique bugs, including 23 in TiDB, 18 in MySQL, and 15 in MariaDB. Among them, 52 bugs were confirmed, 18 fixed, and 6 known.

Bug Severity. Regarding the 23 bugs found in TiDB, two were classified as *Critical* bugs, while 7 as *Major* bugs. The other bugs found in TiDB were assigned low severity (*e.g.*, *Minor*). In MariaDB, all the found bugs were classified as *Critical* (9) or *Major* (6). Most of the bugs reported to the MySQL developers are confidential due to security concerns, so their severity is unavailable. 8 CVEs have been assigned to these security-related bugs. Additionally, 4 bugs posted publicly were classified as *Severe*. These results demonstrate that TxCheck is practical and effective in detecting critical bugs in production DBMSs.

Bug Classification. We classify the 56 bugs according to their root causes. Table 2 shows the results. The class "Transaction" includes the bugs found in transaction executions, and "Non-transaction" the bugs triggered in non-transaction executions, *i.e.*, when the semantically-equivalent test cases are executed (see Section 3.3). The column "Crash" shows the number of bugs that crash DBMS servers, and "Oracle" refers to the bugs that are identified by oracle checking.

In total, TxCheck found 30 bugs triggered in transaction executions, which demonstrates TxCheck's capability for finding real transactional bugs in DBMSs. Among these bugs, 19 were identified by our oracle, and 11 crashed DBMS servers. In addition, 26 bugs were identified in the non-transaction executions, among which 23 crashed DBMS servers and 3

Table 2: Classifying the detected bugs

DBMS	Transaction		Non-transaction	
	Oracle	Crash	Oracle	Crash
TiDB	11	4	1	7
MySQL	4	2	0	12
MariaDB	4	5	2	4
Total	19	11	3	23

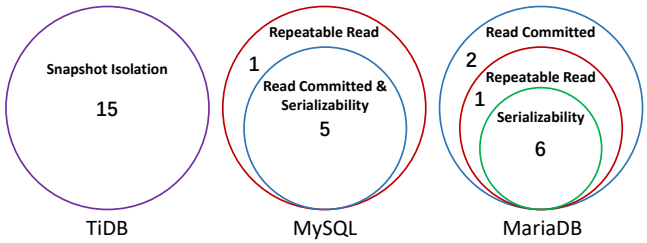

Figure 12: Venn diagrams showing the number of bugs found at different isolation levels. Bugs associated with a smaller circle of isolation level can also be found at larger circles of isolation levels.

were identified by oracle checking. Note that several bugs triggered in non-transaction executions make the execution results incorrect and different from the ones in transaction executions. This demonstrates that TxCheck can also detect incorrect behaviors triggered in non-transaction executions.

Bugs at Different Isolation Levels. For the 30 transactional bugs, Figure 12 shows the isolation levels where they are triggered. All 15 TiDB bugs are identified at *Snapshot Isolation*, which is the only tested isolation level in TiDB. In MySQL, 5 bugs can be found at all three tested isolation levels, while 1 only at *Repeatable Read*. In MariaDB, 6 bugs can be detected at all three tested isolation levels, while 1 at both *Read Committed* and *Repeatable Read*, and 2 only at *Read Committed*. Note that several transactional bugs are independent of isolation guarantees, which can be effectively detected by TxCheck at various levels.

5.2 Comparison with State of the Art

We demonstrate the advantages of our approach by (1) checking whether TxCheck can find new transactional bugs that cannot be found by the state of the art, and (2) discussing selected interesting bugs to show the effectiveness of TxCheck.

For comparison, we analyze the bug-triggering test cases of the 19 transactional bugs found by our oracle checking. We reduce each test case to a minimal bug-inducing version before analysis. Given that there exists no approach for finding general transactional bugs, we select ELLE [4] as competing tool (part of the prevalent testing framework Jepsen [18]). Elle is the state-of-the-art black-box checker for finding isolation bugs, which are a specific kind of transactional bugs.

Table 3: Feature analysis of the 19 bug-triggering test cases

ID	DBMS	Features		ELLE
		Complex	Predicate	
1	TiDB	✓	✓	-
2	TiDB	✓	✓	-
3	TiDB	✓	✓	-
4	TiDB	-	-	✓
5	TiDB	-	-	-
6	TiDB	✓	✓	-
7	TiDB	✓	✓	-
8	TiDB	✓	✓	-
9	TiDB	✓	✓	-
10	TiDB	✓	✓	-
11	TiDB	✓	✓	-
12	MySQL	✓	✓	-
13	MySQL	✓	✓	-
14	MySQL	✓	✓	-
15	MySQL	✓	✓	-
16	MariaDB	✓	✓	-
17	MariaDB	✓	✓	-
18	MariaDB	✓	✓	-
19	MariaDB	✓	✓	-

As shown in Table 3, among the 19 test cases, 17 use both complex statements and predicates. ELLE cannot generate such test cases, because (1) the complex statements do not follow the test-case patterns of ELLE whose write operations can only append; and (2) ELLE does not support predicates. Figure 1 depicts one of such bugs triggered by complex statements and predicates. Regarding the two bugs that do not involve complex operations and predicates, ELLE can find only one of them. We also analyze 11 transactional bugs that crash DBMS servers and find that all of them involve complex statements and predicates, for which ELLE lacks support. In the following, we first illustrate the only bug that can be found by both TxCheck and ELLE. Then, we discuss three representative bugs that are missed by ELLE.

TiDB Bug: Isolation Violation. Figure 13 shows a bug-triggering test case. ELLE can only detect this bug from the 56 bugs found by TxCheck as the corresponding test case does not contain predicates or complex operations. This bug triggers a prohibited phenomenon, *G-SIb*, in Snapshot Isolation [1, 2] used by TiDB. *G-SIb* is an anomaly where the transaction-dependency graph contains a cycle with exactly one anti-dependency edge. To find this bug, TxCheck converts the constructed statement-dependency graph to a transaction-dependency graph (see Section 4) and checks whether there is any prohibited phenomenon. This bug-finding process illustrates that TxCheck can also find isolation bugs.

MySQL Bug: Aborted Transactions Have Effects. Figure 14 shows a test case with two interleaving transactions. Transaction `T1` inserts four items into table `t0` and then rollbacks. Transaction `T0` updates the items that satisfy a complex

Test case

```
T0.S0> start transaction;
T1.S0> start transaction;
T1.S1> select ref0.c0 from t0 as ref0
       order by c0 desc; --- output 1 row
T0.S1> delete from t0; --- delete 1 row
T0.S2> commit;
T1.S2> select ref0.c0 from t0 as ref0
       order by c0 desc; --- output 0 row
T1.S3> commit;
```

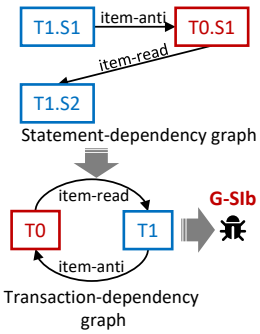

Figure 13: A test case violates *Snapshot* Isolation in TiDB.

Test case	Oracle
T0> start transaction; T1> start transaction; T1> insert into t0 values (141, 210000, ..., 74), ..., (141, 213000, ..., null); T1> rollback; T0> update t0 set vkey = 116 where t0.c5 not in (select subq0.c0 as c0 from (select ...) as subq0 where subq0.c0 < (select ...) order by c0 asc); --- update 39 rows T0> commit;	update t0 set vkey = 116 where t0.c5 not in (select subq0.c0 as c0 from (select ...) as subq0 where subq0.c0 < (select ...) order by c0 asc); --- update 0 row

Figure 14: An aborted transaction affects the results of a committed transaction in MySQL.

predicate (*i.e.*, the **`WHERE`** clause of the **`UPDATE`** statement) from table `t0` and then commits. This test case is semantically-equivalent to executing the **`UPDATE`** statement on the same databases as the aborted transaction must not cause any visible side effects [1, 41]. However, MySQL produces different results: one test case updates 39 rows while the other zero rows. ELLE cannot find this bug as it involves predicates and complex operations, for which ELLE lacks support.

TiDB Bug: Incorrect Transactional Calculation. As shown in Figure 15, the test case contains only one transaction and uses only simple operations without predicates. ELLE can generate such a test case, at least conceptually. However, ELLE cannot find this bug, because it does not violate any isolation specification rather than makes the DBMS return incorrect results. TxCheck finds this bug by constructing semantically-equivalent test cases.

MariaDB Bug: Crash Caused by Transactions. As shown in Figure 16, the test case contains two interleaved transactions. Transaction `T1` first inserts a couple of items into table `t0`. Then, transaction `T0` executes a **`DELETE`** statement with a complex **`WHERE`** clause as its predicate. The deletion is blocked because its predicate matching references certain items of table `t0`, which have just been updated by the **`INSERT`** statement of `T1`. Only after `T1` commits or aborts, can the **`DELETE`** statement be unblocked. While the deletion of `T0` is blocked, transaction `T1` executes a simple **`UPDATE`** statement, which eventually crashes the MariaDB server. This bug is

Test case	Oracle

T0> start transaction;
T0> update t_0 set c_0 = t_2.c_1;
T0> select count(c_2) from t_0;
--- output 39
T0> commit;
	update t_0 set c_0 = t_2.c_1;
	select count(c_2) from t_0;
	--- output 36

Figure 15: A test case makes TiDB return incorrect results.

Test case

```
T0> start transaction;
T1> start transaction;
T1> insert into t0 (vkey, pkey, c0) values
      (89,188000,40), ..., (97, 230000, 9);
T0> delete from t1 where exists (
      select ref0.c0 from t2 as ref0
      where t1.c0 not in (
        select ref3.vkey as c0
        from (t0 as ref2 left outer join t2 as ref3
              on (ref2.vkey = ref3.vkey ))
        where ref3.pkey >= ref2.vkey)); --- blocked
T1> update t2 set vkey = 99; --- crash
```

Figure 16: Two transactions crash the MariaDB server.

due to a concurrency issue where one of the threads performs complex operations that make the DBMS enter erroneous states. ELLE cannot find this bug as it does not support such complex operations involving predicates.

5.3 Design Choice Analysis

We had two considerations while designing our approach. First, can we use existing transaction-dependency graphs, *e.g.*, Directed Serialization Graph (DSG) [1, 2], instead of the proposed statement-dependency graphs? Second, when performing topological sorting, TxCheck randomly selects one node if there are multiple nodes with zero in-degree. Does the random strategy affect the results?

Using Transaction-dependency Graphs. We argue that using transaction-dependency graphs may miss bugs. A transaction-dependency graph in some isolation levels may have cycles as some transactions reference the items that other transactions have referenced. To construct transactional oracles, we must refine a test case to ensure the acyclicity of the associated graph. However, a test case may have an acyclic statement-dependency graph, but a cyclic transaction-dependency graph. Such test cases are unable to be topologically sorted at the transaction level, and thus interesting test cases may be discarded.

To demonstrate that using transaction-dependency graphs may miss bugs, we check the transaction-dependency graphs of the 19 transactional bugs found by our oracle checking. We first check the graphs built on minimized test cases and find that all the constructed transaction-dependency graphs miss cycles. It is unsurprising as, when minimizing the test cases, we delete all the unnecessary clauses and statements,

Table 4: Analysis of transaction-dependency graphs

DBMS	Test cases	Txn-cycle
TiDB	11	3
MySQL	4	1
MariaDB	4	2
Total	19	6

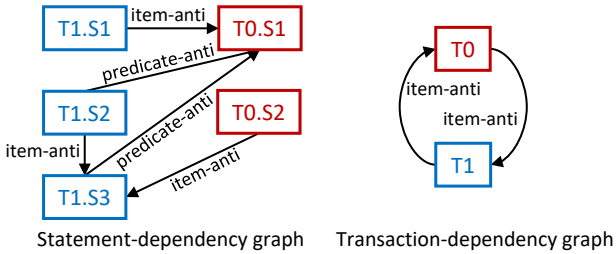

Figure 17: The statement-dependency and transaction-dependency graphs of the test case in Figure 1.

which makes the test cases reference much fewer items than the non-minimized ones. However, randomly generated test cases inevitably contain many redundant parts [28, 34]. To understand whether the test cases generated by TxCheck contain cycles in transaction-dependency graphs, we check the non-minimized test cases accordingly. Table 4 shows the results. The column *Txn-cycle* refers to the number of test cases that have cycles in transaction-dependency graphs.

The results show that 6 bug-triggering test cases have cycles in transaction-dependency graphs. If we use transaction-dependency graphs instead of statement-dependency graphs, these test cases are not suitable for constructing oracles as topological sorting cannot be performed for cyclic graphs. Hence, around one-third (6 out of 19) of the bugs would be missed. The test case in Figure 1 exemplifies such a bug. Figure 17 presents its corresponding statement-dependency and transaction-dependency graphs, respectively. The statement-dependency graph does not have cycles, so topological sorting can be performed at the statement level, which reveals the bug. Topological sorting is infeasible at the transaction level as the transaction-dependency graph is cyclic.

Random Topological Sorting. TxCheck topologically sorts statement-dependency graphs to construct oracles. If TxCheck encounters multiple nodes whose in-degrees are zero during sorting, it randomly selects one of them. In this way, TxCheck chooses only one of the topological sorting results to construct the oracle. For any correct DBMS, all the topological sorting results must be the same as the transaction execution results according to Theorem 1. However, when bugs are triggered, the test cases executed with transactions and some of the sorted test cases may produce the same, yet incorrect results. If TxCheck, unfortunately, chooses such sorted test cases, bugs may be overlooked.

Table 5: Analysis of topological sorting: ⋆ labels the test cases that generate millions of topological sorting results, where we randomly select 10k to check whether they can trigger bugs

ID	DBMS	Sort	Trigger	ID	DBMS	Sort	Trigger
1	TiDB	1	1 (100%)	11	TiDB	1260	1224 (97%)
2	TiDB	32	32 (100%)	12	MySQL	6	6 (100%)
3	TiDB	12	12 (100%)	13	MySQL	1	1 (100%)
4	TiDB	-	-	14	MySQL	1	1 (100%)
5	TiDB	1	1 (100%)	15	MySQL	10k⋆	10k (100%)
6	TiDB	6	6 (100%)	16	MariaDB	2	2 (100%)
7	TiDB	6	6 (100%)	17	MariaDB	1	1 (100%)
8	TiDB	30	30 (100%)	18	MariaDB	1	1 (100%)
9	TiDB	96	96 (100%)	19	MariaDB	10k⋆	10k (100%)
10	TiDB	36	36 (100%)				

We show that such missed bugs are rare in practice. Typically, transactional bugs affect the results of transaction executions, while non-transaction executions of the topologically sorted test cases would not be affected. Therefore, most sorted test cases should execute correctly and can be used as oracles to reveal bugs. To demonstrate this, we analyze the 19 transactional bugs found by our oracle checking. The analyzed test cases are not minimized as we intend to obtain results that are close to those from the generated test cases in practice. Table 5 shows the results. The column *Sort* shows the number of all possible topological sorting results for the statement-dependency graphs. The column *Trigger* refers to the number of sorting results that successfully trigger the bugs.

Among the 19 test cases, the sorting results of 17 can stably trigger the bugs with 100% success rates. Bug 4 is found by checking transaction-dependency graphs, which do not involve topological sorting, as discussed in Section 5.2. The test case for bug 11 produces 1260 possible sorting results, among which 1224 can trigger the bug. It indicates that missed bugs can indeed happen with, however, a low probability (less than 3%). Two bug-triggering test cases (bugs 15 and 19) generate millions of sorting results. Both of them contain dozens of statements and involve few dependencies. When the dependency constraint is weak, the number of possible topological sorting will explode (*e.g.*, without any dependency, 12 statements can already amount to 12!, *i.e.*, over 480 million, sorting results). We randomly select 10K sorting results for each test case and find that all of them can trigger the bugs. These results show that randomly selecting one topological sorting result for oracle checking is practical and effective.

6 Discussion

Test-case Generation. TxCheck generates databases and transactions randomly for testing. Such random generation may be inefficient to explore corner test cases and thus may miss bugs. Fuzzing is a promising technique for generating infrequently executed test cases [3, 10, 16, 19, 27], and has been adopted in DBMS testing [21, 50, 53]. However, traditional fuzzing techniques cannot be directly utilized in DBMS

transaction testing. First, code coverage, which is commonly used as the fuzzing feedback, is not well suited because it cannot measure transaction interleavings. Second, random mutations used by most fuzzers are ineffective for generating transactions with complex data dependencies. One promising approach to addressing these challenges is to design new coverage feedback and mutation strategies following work on concurrency fuzzing [12, 20, 51].

Predicate Handling. It is challenging to recover transaction histories involving predicates [4, 44], for which we provide a possible solution. We instrument *Version-Set Read* (*VSR*) statements to capture items referenced by predicates by counting all items in the referenced tables. However, as discussed in Section 3.2, this method may overcount the referenced items because it is unnecessary that all items in the referenced tables would be referenced. Therefore, TxCheck may build spurious dependencies between statements. To mitigate this issue, one may utilize domain-specific knowledge. For example, we can customize *VSR* statements for each specific kind of statement used in transactions by referring to the corresponding SQL grammars and features.

Data-intensive Transaction. Existing work [21, 22, 35–37, 53] demonstrates that many DBMS bugs can be triggered without using much data. We follow this insight; each database generated by TxCheck generally contains 50-80 rows of data. However, some transactional bugs may hide in code only reached when intensive data is processed. To find such bugs, we plan to enable TxCheck to generate databases with large amounts of data. However, as TxCheck needs to reset database states after each test, its performance will be significantly degraded when TxCheck resets complicated databases. We plan to experiment with snapshot techniques to help improve testing performance by following and adapting existing work [24, 38].

7 Related Work

Transaction Testing. Transaction-testing approaches validate the correct uses of transactions in applications [14, 15] or the correct implementations of transaction support of DBMSs [4, 9, 11, 17, 44]. AGENDA [14, 15] tests DB-based applications that utilize transactions to perform certain tasks. It generates test cases according to user-provided specifications. A bug is reported if the application incorrectly constructs transactions that violate the provided specifications. The black-box isolation checkers ELLE [4], COBRA [44], and POLYSI [17] examine whether the transaction support of DBMSs functions correctly. ELLE generates transactions that use "append" operations as writes and can naturally recover their version order according to the list of values. COBRA and POLYSI focus respectively on validating the *Serializability* and *Snapshot Isolation* guarantees of transactions in DBMSs and develop several techniques (*e.g.*, read-modify-write transaction-based version order inferring, compact constraint encoding for SMT solving, and parallel hardware) to enable fast dependency inference. Unlike existing checkers, TxCheck focuses on testing the transaction support of DBMSs while relaxing the constraints on test-case patterns and enabling complex transaction generation. Moreover, TxCheck provides a practical solution to inferring predicate dependencies, a challenging problem in DBMS transaction testing.

DBMS Testing. Automated testing approaches have been proposed to find other types of bugs in DBMSs, such as logic bugs [35–37, 52], security bugs [21, 43, 50, 53], and performance bugs [22, 26]. SQLancer [42] is a well-known DBMS testing tool for detecting logic bugs, which integrates several approaches [35–37]. PQS [37] constructs queries that require DBMSs to return target items from manipulated databases: a logic bug is reported if the tested DBMS fails to return such items. TLP [36] designs some patterns to partition an original query into three separate queries, so that the union of the separated queries' results must be the same as the original query's result; otherwise, TLP reports a bug. Focusing on memory bugs, both SQUIRREL [53] and DynSQL [21] can generate more diverse test cases. SQUIRREL utilizes intermediate representations to model the structures of queries and the dependencies between statements. This enables SQUIRREL to generate queries containing multiple statements. By merging the query generation and query processing, DynSQL incrementally generates complex and valid queries using the state information of DBMSs. We also design TxCheck for tackling the oracle problem. However, TxCheck focuses on bugs triggered in transactional scenarios. In addition, with moderate test-case pattern constraints, TxCheck can handle complex test cases and expose deep transactional bugs.

8 Conclusion

We have presented a novel DBMS transaction testing approach, along with the practical tool TxCheck. Our approach is based on statement-level dependency graphs and can generate semantically-equivalent test cases to validate the transaction executions. TxCheck has found 56 unique bugs in three widely-used DBMSs, among which 52 have been confirmed and 18 fixed. Thanks to its generality and effectiveness, we expect TxCheck to help developers design and implement correct and reliable DBMS transaction support. Moreover, our approach could be utilized to infer predicate-related dependencies in recovering transaction histories.

Acknowledgments

We thank the anonymous OSDI reviewers and our shepherd, Tianyin Xu, for their valuable feedback on earlier versions of this paper. We also thank the DBMS developers for triaging and fixing our reported bugs.

References

[1] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[2] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, pages 67–78, 2000.

[3] American fuzzy lop. https://github.com/google/AFL.

[4] Peter Alvaro and Kyle Kingsbury. Elle: Inferring isolation anomalies from experimental observations. In *Proceedings of the 46th International Conference on Very Large Databases (VLDB)*, pages 268–280, 2020.

[5] Peter Bailis, Aaron Davidson, Alan D. Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. In *Proceedings of the 39th International Conference on Very Large Databases (VLDB)*, pages 181–192, 2013.

[6] Peter Bailis, Alan D. Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. Scalable atomic visibility with RAMP transactions. In *Proceedings of the 2014 International Conference on Management of Data (SIGMOD)*, pages 27–38, 2014.

[7] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. Strong consistency is not hard to get: Two-phase locking and two-phase commit on thousands of cores. In *Proceedings of the 45th International Conference on Very Large Databases (VLDB)*, pages 2325–2338, 2019.

[8] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control - theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[9] Ranadeep Biswas and Constantin Enea. On the complexity of checking transactional consistency. In *Proceedings of the 2019 International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 165:1–165:28, 2019.

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 23rd International Conference on Computer and Communications Security (CCS)*, pages 1032–1043, 2016.

[11] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In *Proceedings of the 26th International Conference on Concurrency Theory*, pages 58–71, 2015.

[12] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. MUZZ: thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *Proceedings of the 29th USENIX Security Symposium*, pages 2325–2342, 2020.

[13] DB-Engines Ranking, Accessed in May, 2023. https://db-engines.com/en/ranking.

[14] Yuetang Deng, Phyllis G. Frankl, and David Chays. Testing database transactions with AGENDA. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 78–87, 2005.

[15] Yuetang Deng, Phyllis G. Frankl, and Zhongqiang Chen. Testing database transaction concurrency. In *Proceedings of the 18th International Conference on Automated Software Engineering (ASE)*, pages 184–195, 2003.

[16] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: data flow sensitive fuzzing. In *Proceedings of the 29th USENIX Security Symposium*, pages 2577–2594, 2020.

[17] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David A. Basin, Haixiang Li, and Anqun Pan. Efficient black-box checking of snapshot isolation in databases. *Proc. VLDB Endow.*, 16(6):1264–1276, 2023.

[18] Jepsen. https://jepsen.io/.

[19] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Fuzzing error handling code using context-sensitive software fault injection. In *Proceedings of the 29th USENIX Security Symposium*, pages 2595–2612, 2020.

[20] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*, 2022.

[21] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. DynSQL: Stateful fuzzing for database management systems with complex and valid sql query generation. In *Proceedings of the 32nd USENIX Security Symposium*.

[22] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. APOLLO: automatic detection and diagnosis of performance regressions in database systems. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, pages 57–70, 2019.

[23] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. Query optimization by predicate move-around. In *Proceedings of the 20th International Conference on Very Large DataBases (VLDB)*, pages 96–107, 1994.

[24] Junqiang Li, Senyi Li, Gang Sun, Ting Chen, and Hongfang Yu. Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots. *IEEE Transactions on Information Forensics and Security*, 17:2673–2687, 2022.

[25] libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[26] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. Testing dbms performance with mutations. *arXiv preprint arXiv:2105.10016*, 2021.

[27] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Security Symposium*, pages 1949–1966, 2019.

[28] David Maciver and Alastair F. Donaldson. Test-case reduction via test-case generation: Insights from the hypothesis reducer. In Robert Hirschfeld and Tobias Pape, editors, *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP)*, volume 166, pages 13:1–13:27, 2020.

[29] MariaDB. https://www.mariadb.org/.

[30] MySQL. https://www.mysql.com/.

[31] MySQL Transaction Isolation Levels. https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html.

[32] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 International Conference on Management of Data (SIGMOD)*, pages 677–689, 2015.

[33] Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the Association for Computing Machinery (JACM)*, 26(4):631–653, 1979.

[34] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 2012 International Conference on Programming Language Design and Implementation (PLDI)*, pages 335–346, 2012.

[35] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESE/FSE)*, pages 1140–1152, 2020.

[36] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. In *Proceedings of the 2020 International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 1–30, 2020.

[37] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 667–682, 2020.

[38] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *Proceedings of the 30th USENIX Security Symposium*, pages 2597–2614, 2021.

[39] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition.* Addison-Wesley, 2011.

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 309–318, 2012.

[41] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020.

[42] SQLancer. https://github.com/sqlancer/sqlancer.

[43] SQLsmith: a random sql query generator. https://github.com/anse1/sqlsmith.

[44] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. Cobra: Making transactional key-value stores verifiably serializable. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 63–80, 2020.

[45] Alexander Thomasian. Two-phase locking performance and its thrashing behavior. *ACM Transactions on Database Systems*, 18(4):579–625, 1993.

[46] TiDB. https://www.pingcap.com/tidb/.

[47] Tidb pessimistic transaction mode. https://docs.pingcap.com/tidb/stable/pessimistic-transaction.

[48] Tidb transaction isolation levels. https://docs.pingcap.com/tidb/stable/transaction-isolation-levels.

[49] What is a Transaction? http://msdn.microsoft.com/en-us/library/aa366402(VS.85).aspx.

[50] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. Industry practice of coverage-guided enterprise-level DBMS fuzzing. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE SEIP)*, pages 328–337, 2021.

[51] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. KRACE: data race fuzzing for kernel file systems. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, pages 1643–1660, 2020.

[52] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. Finding bugs in gremlin-based graph database systems via randomized differential testing. In *ISSTA'22*, pages 302–313. ACM, 2022.

[53] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 International Conference on Computer and Communications Security (CCS)*, pages 955–970, 2020.

A Definition for statement dependencies

We define statement dependencies in the same fashion as for the transaction dependencies defined by Adya *et al.* [1, 2].

Definition 2 (Directly stmt-item-read-depends) Statement S_j directly stmt-item-read-depends on statement S_i if S_i installs an item version x_i while S_j reads x_i.

Definition 3 (Directly stmt-item-write-depends) Statement S_j directly stmt-item-write-depends on statement S_i if S_i installs an item version x_i while S_j installs x's next version (after x_i) in the version order.

Definition 4 (Directly stmt-item-anti-depends) Statement S_j directly stmt-item-anti-depends on statement S_i if S_i reads an item version x_k while S_j installs x's next version (after x_k) in the version order.

Definition 5 (Directly stmt-predicate-read-depends) Statement S_j directly stmt-predicate-read-depends on statement S_i if S_j performs an operation r_j(P: Vset(P)) while S_i installs an item version x_i that is included in Vset(P).

Definition 6 (Directly stmt-predicate-write-depends) Statement S_j directly stmt-predicate-write-depends on statement S_i if either (1) S_j overwrites an operation w_i(P: Vset(P)) performed by S_i, or (2) S_j executes an operation w_j(Q: Vset(Q)) while S_i installs an item version x_i that is included in Vset(Q).

Definition 7 (Directly stmt-predicate-anti-depends) Statement S_j directly stmt-predicate-anti-depends on statement S_i if S_j overwrites an operation r_i(P: Vset(P)) performed by S_i.

Definition 8 (Directly stmt-value-write-depends) Statement S_j directly stmt-value-write-depends on statement S_i if either (1) S_i executes an operation w_i^{value}(E: Vset(E)) where x_k is included while S_j install x's next version (after x_k) in version order, or (2) S_j executes an operation w_j^{value}(F: Vset(F)) while S_i installs x_i that is included in Vset(F). Here, statement S_i performs w_i^{value}(E: Vset(E)) if S_i installs an item version whose values are based on expression E and the system (conceptually) reads all needed versions in Vset(E).

B Proof related to SQL-level instrumentation

Assumption 1 prohibits statements use old item versions while the newer ones are conceptually available. This can happen in distributed DBMSs when a new item version is produced but not well-synchronized, and thus the old version is still used in some machines. However, this work focuses on bugs in database engines deployed in local machines. Therefore, it is reasonable to assume that every item version is well-synchronized. Assumption 2 ensures that the inserted VSR statements can correctly work. Assumption 3 ensures that the inserted BWR statements read the same item version used in the target statements.

Lemma 1 Statement S_j directly stmt-item-read-depends on statement $S_i \Leftrightarrow$ outputs of the AWR statement of S_i and outputs of S_j have intersections.

Proof: (1) Statement S_j directly stmt-item-read-depends on statement $S_i \Rightarrow$ outputs of the AWR statement of S_i and outputs of S_j have intersections. Because S_i directly write-read depends on S_j, there is an item x such that S_i installs version x_i and S_j reads x_i. S_i installs version x_i, so x_i must be included in the output of AWR of S_j. S_j reads x_i, so x_i must be in the output of S_j. So the output of AWR of S_i and the output of S_j have intersections.

(2) Statement S_j directly stmt-item-read-depends on statement $S_i \Leftarrow$ outputs of the AWR statement of S_i and outputs of S_j have intersections. Suppose x_k is one of the intersected item versions. x_k is in the output of AWR of S_i, so x_k must be installed by S_i because S_i is the only one statement that can assign the corresponding *VersionKey* value to x_k that matches the predicates of AWR of S_i. x_k is also in the output of S_j, so x_k is read by S_j. So S_i installs an item version x_k and S_j reads x_k. Therefore, S_i write-read depends on S_j.

Combining (1) and (2), we prove Lemma 1.

Lemma 2 Statement S_j directly stmt-item-write-depends on statement $S_i \Leftrightarrow$ outputs of the AWR statement of S_i and outputs of the BWR statement of S_j have intersections.

Proof: (1) Statement S_j directly stmt-item-write-depends on statement $S_i \Rightarrow$ outputs of the AWR statement of S_i and outputs of the BWR statement of S_j have intersections. Because S_j directly write-write depends on S_i, S_i installs a version x_i and S_j installs x's next version (after x_i) in the version order. S_i installs a version x_i, so x_i must be in the output of AWR of S_i. Suppose x_k is the version of item x that is used in the predicate matching of S_j. x_k must satisfy the predicate of S_j because S_j is going to install a new version for item x. As BWR of S_j uses the same predicate as S_j, x_k must be in the output set of BWR of S_j. Suppose S_j is going to install x_j, there must be $x_k \ll x_j$. And x_j is x_i's next version, so x_k is x_i, or $x_k \ll x_i$.

We assume that $x_k \ll x_i$. (a) If S_i and S_j are in the same transaction. x_i must be installed before x_j, which is installed by S_j, and $x_k \ll x_i$. Therefore, the BWR of S_j must read version x_i instead of x_k. Conflict. (b) If S_i and S_j are in different transactions, T_i and T_j. Because T_j reads x_k, and T_i installs x_i, which is after x_k, T_i item-anti-depends on T_j. Because T_i installs x_i, and T_j installs x_j that are after x_i, T_j item-write-depends on T_i. Conflict with Assumption 3.

Combining (a) (b), we get $x_k \ll x_i$ in conflict. So x_k is x_i. So x_i is in the output of BWR of S_j. And x_i is in the output of

AWR of S_i. The output of AWR of S_i and the output of BWR of S_j have intersections.

(2) Statement S_j directly stmt-item-write-depends on statement $S_i \Leftarrow$ outputs of the AWR statement of S_i and outputs of the BWR statement of S_j have intersections. Suppose x_k is one of the intersected item versions. x_k is in the output of AWR of S_i, so x_k must be installed by S_i because S_i is the only one statement that can assign the corresponding *VersionKey* value to x_k that matches the predicates of AWR of S_i. x_k is in the output of BWR of S_j, so x_k satisfies the predicate of BWR of S_j, and it satisfies the predicate of S_j. Therefore, S_j will install a new version of x. So S_i installs a version x_k and S_j installs x's next version (after x_k) in the version order, which means that S_j directly write-write depends on S_i.

Combining (1) and (2), we prove Lemma 2.

Lemma 3 Statement S_j directly stmt-item-anti-depends on statement $S_i \Leftrightarrow$ outputs of S_i and outputs of the BWR statement of S_j have intersections.

Proof: (1) Statement S_j directly stmt-item-anti-depends on statement $S_i \Rightarrow$ outputs of S_i and outputs of the BWR statement of S_j have intersections. Because S_i directly stmt-item-anti depends on S_j, S_i reads an item version x_k and S_j installs x's next version (after x_k) in the version order. Suppose x_h is the version of item x that is used in the predicate matching of S_j. x_h must satisfy the predicate of S_j because S_j is going to install a new version for item x. As BWR of S_j uses the same predicate as S_j, x_h must be in the output of BWR of S_j. Suppose S_j is going to install x_j, there must be $x_h \ll x_j$. While x_j is x_k's next version, so x_k is x_h, or $x_h \ll x_k$.

We assume that $x_h \ll x_k$: (a) S_i and S_j are in the same transaction. S_i must be before S_j. S_i reads version x_k, and thus S_j must use a version of x that is after x_k or equal to x_k (Assumption 1). However, S_j uses version x_h, and $x_h \ll x_k$. Conflict. (b) S_i and S_j are in different transactions, T_i and T_j. Suppose statement S_k of transaction T_k installs version x_k. If T_k is T_j, S_k must before S_j because S_j installs x_k's next version. So x_k is visible to S_j, and S_j must use x_k instead of x_h ($x_h \ll x_k$). Conflict. So T_k is not T_j. Because T_j reads x_h, and T_k installs x_k that are after x_h, so T_k item-anti-depends on T_j. And T_j installs x_j that are after x_k, so T_j item-write-depends on T_k. Conflict with Assumption 3.

Combining (a) (b), we get $x_h \ll x_k$ in conflict. So x_k is x_h. So x_k is in the output of BWR of S_j. And x_k is in the output of S_i, The output of S_i and the output of BWR of S_j have intersections.

(2) Statement S_j directly stmt-item-anti-depends on statement $S_i \Leftarrow$ outputs of S_i and outputs of the BWR statement of S_j have intersections. Suppose x_k is one of the intersected items. x_k is in the output of BWR of S_j, so x_k satisfies the predicate of BWR of S_j, and thus it satisfies the predicate of S_j (S_j and BWR of S_j use same version of x). Therefore, S_j will install a version after x_k. x_k is in the output of S_i, so x_k is read by S_i. So S_i reads a item version x_k and S_j installs x with a version after x_k in the version order. Therefore, S_i directly stmt-item-anti-depends on S_j.

Combining (1) and (2), we prove Lemma 3.

Lemma 4 Statement S_j directly stmt-predicate-read-depends on statement $S_i \Rightarrow$ outputs of the AWR statement of S_i and outputs of one of the VSR statements of S_j have intersections.

Proof: Statement S_j directly stmt-predicate-read-depends on statement S_i, which means that S_j performs an operation r_j(P: Vset(P)), and there is an item version x_i that is installed by S_i and $x_i \in$ Vset(P). Because x_i is installed by S_i, the x_i must be included in the output of AWR of S_i. Because VSRs of S_j outputs all item versions in the referenced tables, according to Assumption 2, all referenced item versions should be outputted by VSRs of S_j. Because $x_i \in$ Vset(P), at least one of the VSRs of S_j outputs x_i. So outputs of the AWR statement of S_i and outputs of one of the VSR statements of S_j have intersections. Proved.

Lemma 5 Statement S_j directly stmt-predicate-write-depends on statement $S_i \Rightarrow$ (1) outputs of one of the VSR statements of S_i and outputs of the BWR statement of S_j have intersections, or (2) outputs of the AWR statement of S_i and outputs of one of the VSR statements of S_j have intersections.

Proof: If statement S_j directly stmt-predicate-write-depends on statement S_i, according to Definition 6, it can be (1) S_j overwrites an operation w_i(P: Vset(P)) performed by S_i, or (2) S_j executes an operation w_j(Q: Vset(Q)) while S_i installs an item version x_i that is included in Vset(Q).

For case (1), S_j overwrites operation w_i(P: Vset(P)), which means that S_i performs an operation w_i(P: Vset(P)), and there exists $x_k \in$ Vset(P) that S_j installs x's next version (after x_k). Because $x_k \in$ Vset(P), x_k must be in the output of one of the VSRs of S_i. Suppose x_h is the version of x that is used in S_j for predicate matching. Because S_j installs x's next version (after x_k), x_h must satisfy the predicate of S_j because S_j is going to install a new version for item x. As BWR of S_j uses the same predicate as S_j, x_h must be in the output of BWR of S_j too. Suppose S_j is going to install x_j, there must be $x_h \ll x_j$. And x_j is x_k's next version, so x_k is x_h, or $x_h \ll x_k$.

We assume that $x_h \ll x_k$: (a) S_i and S_j are in the same transaction. S_i must be before S_j. S_i uses version x_k, and thus S_j must use a version of x that is later than or equal to x_k. However, S_j uses x_h and $x_h \ll x_k$. Conflict. (b) S_i and S_j are in different transactions, T_i and T_j. Suppose statement S_k of transaction T_k installs version x_k. If T_k is T_j, S_k must be before S_j because S_j install x_k's next version. So x_k is visible to S_j, so S_j must use x_k instead of x_h as $x_h \ll x_k$. Conflict. So T_k is not T_j. Because T_j reads x_h (BWR of S_j) while T_k installs x_k that are after x_h, T_k item-anti-depends on T_j. Because T_j installs x_j that are after x_k, T_j item-write-depends on Conflict with Assumption 3.

Combining (a) (b), we get $x_h \ll x_k$ in conflict. So x_k is x_h. So x_k is in the output of BWR of S_j. And x_k is in the output of one of the VSRs of S_i, the outputs of one of the VSR statements of S_i and the output of BWR of S_j have intersections.

For case (2), S_j executes an operation w_j(Q: Vset(Q)) while S_i installs an item version x_i that is included in Vset(Q). Because x_i is installed by S_i, the x_i must be included in the output of AWR of S_i. Because $x_i \in$ Vset(Q), x_i must be in the output of one of the VSRs of S_j. So the output of AWR of S_i and the output of one of the VSRs of S_j have intersections.

Combining (1) and (2), we prove Lemma 5.

Lemma 6 Statement S_j directly stmt-predicate-anti-depends on statement $S_i \Rightarrow$ outputs of one of the VSR statements of S_i and outputs of the BWR statement of S_j have intersections.

Proof: Statement S_j directly stmt-predicate-anti-depends on statement S_i, which means that S_i performs an operation r_i(P: Vset(P)), and there exists $x_k \in$ Vset(P) that S_j installs x's next version (after x_k). Because $x_k \in$ Vset(P), x_k must be in the output of one of the VSRs of S_i. Suppose x_h is the version of x that is used in S_j for predicate matching. Because S_j installs x's next version (after x_k), x_h must satisfy the predicate of S_j because S_j is going to install a new version for item x. As BWR of S_j uses the same predicate as S_j, x_h must be in the output of BWR of S_j too. Suppose S_j is going to install x_j, there must be $x_h \ll x_j$. And x_j is x_k's next version, so x_k is x_h, or $x_h \ll x_k$.

We assume that $x_h \ll x_k$: (a) S_i and S_j are in the same transaction. S_i must be before S_j. S_i uses version x_k, and thus S_j must use a version of x that is later than or equal to x_k. However, S_j uses x_h and $x_h \ll x_k$. Conflict. (b) S_i and S_j are in different transactions, T_i and T_j. Suppose statement S_k of transaction T_k installs version x_k. If T_k is T_j, S_k must be before S_j because S_j install x_k's next version. So x_k is visible to S_j, so S_j must use x_k instead of x_h as $x_h \ll x_k$. Conflict. So T_k is not T_j. Because T_j reads x_h (BWR of S_j) while T_k installs x_k that are after x_h, T_k item-anti-depends on T_j. Because T_j installs x_j that are after x_k, T_j item-write-depends on Conflict with Assumption 3.

Combining (a) (b), we get $x_h \ll x_k$ in conflict. So x_k is x_h. So x_k is in the output of BWR of S_j. And x_k is in the output of one of the VSRs of S_i, the outputs of one of the VSR statements of S_i and the output of BWR of S_j have intersections. Lemma 6 is proved.

Lemma 7 Statement S_j directly stmt-value-write-depends on statement $S_i \Rightarrow$ (1) outputs of one of the VSR statements of S_i and outputs of the BWR statement of S_j have overlapping parts, or (2) outputs of the AWR statement of S_i and outputs of one of the VSR statements of S_j have intersections.

Proof: If statement S_j directly stmt-value-write-depends on statement S_i, according to Definition 8, it can be (1) S_i executes an operation w_i^{value}(E: Vset(E)) where x_k is included while S_j installs x's next version (after x_k) in version order, or (2) S_j executes an operation w_j^{value}(F: Vset(F)) while S_i installs x_i that is included in Vset(F).

For case (1), because $x_k \in$ Vset(E), x_k must be in the output of one of the VSRs of S_i. Suppose x_h is the version of x that is used in S_j for predicate matching. Because S_j installs x's next version (after x_k), x_h must satisfy the predicate of S_j because S_j is going to install a new version for item x. As BWR of S_j uses the same predicate as S_j, x_h must be in the output of BWR of S_j too. Suppose S_j is going to install x_j, there must be $x_h \ll x_j$. And x_j is x_k's next version, so x_k is x_h, or $x_h \ll x_k$.

We assume that $x_h \ll x_k$: (a) S_i and S_j are in the same transaction. S_i must be before S_j. S_i uses version x_k, and thus S_j must use a version of x that is later than or equal to x_k. However, S_j uses x_h and $x_h \ll x_k$. Conflict. (b) S_i and S_j are in different transactions, T_i and T_j. Suppose statement S_k of transaction T_k installs version x_k. If T_k is T_j, S_k must be before S_j because S_j install x_k's next version. So x_k is visible to S_j, so S_j must use x_k instead of x_h as $x_h \ll x_k$. Conflict. So T_k is not T_j. Because T_j reads x_h (BWR of S_j) while T_k installs x_k that are after x_h, T_k item-anti-depends on T_j. Because T_j installs x_j that are after x_k, T_j item-write-depends on Conflict with Assumption 3.

Combining (a) (b), we get $x_h \ll x_k$ in conflict. So x_k is x_h. So x_k is in the output of BWR of S_j. And x_k is in the output of one of the VSRs of S_i, the outputs of one of the VSR statements of S_i and the output of BWR of S_j have intersections.

For case (2), because x_i is installed by S_i, the x_i must be included in the output of AWR of S_i. Because $x_i \in$ Vset(F), x_i must be in the output of one of the VSRs of S_j. So the output of AWR of S_i and the output of one of the VSRs of S_j have intersections.

Combining (1) and (2), we prove Lemma 7.

C Proof for Theorem 1

Inductive proof: n is the number of statements in the statement-dependency graph (SDG). [$T_{x1}S_{y1}$, $T_{x2}S_{y2}$, ... , $T_{xn}S_{yn}$] is the statement sequence executed within transactions. [S_{z1}, S_{z2}, ..., S_{zn}] is the statement sequence generated by performing topological sorting on SDG.

when n = 1, obviously T_1S_1 and S_1 give the same results.

Suppose n = k, its SDG G_k is acyclic, and [$T_{x1}S_{y1}$, $T_{x2}S_{y2}$, ... , $T_{xk}S_{yk}$] and [S_{z1}, S_{z2}, ..., S_{zk}] produce the same results. When n = k + 1, we add a new statement at the end of the transactional statement sequence. Therefore, the sequence becomes [$T_{x1}S_{y1}$, $T_{x2}S_{y2}$, ... , $T_{xk}S_{yk}$, $T_{x(k+1)}S_{y(k+1)}$]. We need to prove that Theorem 1 holds for k+1 if the SDG is acyclic.

Because $T_{x1}S_{y1}$, $T_{x2}S_{y2}$, ... , and $T_{xk}S_{yk}$ are executed before $T_{x(k+1)}S_{y(k+1)}$, they are not affected by $T_{x(k+1)}S_{y(k+1)}$, and thus their execution results are the same as [$T_{x1}S_{y1}$, $T_{x2}S_{y2}$, ... , $T_{xk}S_{yk}$] in the k-length case. So $T_{x1}S_{y1}$, $T_{x2}S_{y2}$, ... , and $T_{xk}S_{yk}$ still generate graph G_k. And then $T_{x(k+1)}S_{y(k+1)}$

is executed and produces some dependencies to some of the executed statements. Therefore, G_{k+1} is a super graph of G_k. By performing topological sort, it generates several normal execution sequences. Suppose [S_{w1}, S_{w2}, ..., S_{wk}, $S_{w(k+1)}$] is one of the sequences. Taking out the new statement $S_{y(k+1)}$ from this sequence, we can get [S_{z1}, S_{z2}, ..., S_{zk}]. Because we use topological sort, and G_{k+1} is a super graph of G_k, [S_{z1}, S_{z2}, ..., S_{zk}] follows the edges in G_{k+1} and thus follows the edges in G_k. Therefore, [S_{z1}, S_{z2}, ..., S_{zk}] must be one of the topologically sorting results of k-length cases.

Now we consider the new statement $S_{y(k+1)}$. Suppose [S_{z1}, S_{z2}, ..., S_{zp}, $S_{y(k+1)}$, $S_{z(p+1)}$, ..., S_{zk}] is one of the topologically sorting results of k+1-length cases. [S_{z1}, S_{z2}, ..., S_{zk}] is the topologically sorting results of k-length cases.

$S_{y(k+1)}$ will not affect the statements that are executed before it. Therefore, the S_{z1}, S_{z2}, ..., S_{zp} produce the same results as they are in transaction execution (according to the k-length case). — (**Conclusion 1**)

Now, we need to prove: (1) $S_{y(k+1)}$ produce the same results as $T_{x(k+1)}S_{y(k+1)}$; and (2) $S_{z(p+1)}$, ..., S_{zk} are not affected by $S_{y(k+1)}$, that is, they will produce the same results as they produce in transaction execution.

(1) $S_{y(k+1)}$ produce the same results as $T_{x(k+1)}S_{y(k+1)}$.

Proof: If $S_{y(k+1)}$ and $T_{x(k+1)}S_{y(k+1)}$ produce different results, there must be at least one item x whose version x_i is referenced by $S_{y(k+1)}$ and x_j is referenced by $T_{x(k+1)}S_{y(k+1)}$, and x_i is different from x_j. There are only two possible cases:

(a) $x_i \ll x_j$. Suppose $T_{xj}S_{yj}$ installs item version x_j, so $T_{x(k+1)}S_{y(k+1)}$ depends on $T_{xj}S_{yj}$ because $T_{x(k+1)}S_{y(k+1)}$ references the item version installed by $T_{xj}S_{yj}$. Therefore, topological sorting will put $T_{xj}S_{yj}$ before $T_{x(k+1)}S_{y(k+1)}$. Suppose S_{zj} is the statement in sorted sequence corresponding to $T_{xj}S_{yj}$. S_{zj} is before $S_{y(k+1)}$, so S_{zj} should produce the same results as $T_{xj}S_{yj}$ according to Conclusion 1. So S_{zj} also installs version x_j. So $S_{y(k+1)}$ must reference the version of item x later than or equal to x_j. However, $S_{y(k+1)}$ reference x_i, and $x_i \ll x_j$. Conflict.

(b) $x_i \gg x_j$. Suppose x_i is installed by S_{zi}. Because $S_{y(k+1)}$ reference x_i, S_{zi} must be before $S_{y(k+1)}$. According to Conclusion 1, S_{zi} produces the same results as it is in transaction execution. Suppose $T_{xi}S_{yi}$ is the corresponding statement in the transaction execution. $T_{xi}S_{yi}$ installs item version x_i while $T_{x(k+1)}S_{y(k+1)}$ reference x_j that is older than x_i, so $T_{xi}S_{yi}$ depends on $T_{x(k+1)}S_{y(k+1)}$. Therefore, topological sorting will put $T_{xi}S_{yi}$ after $T_{x(k+1)}S_{y(k+1)}$, *i.e.*, S_{zi} is after $S_{y(k+1)}$, which is in conflict with that S_{zi} must be before $S_{y(k+1)}$.

Combining (a) and (b), we can get that there is no item that $S_{y(k+1)}$ and $T_{x(k+1)}S_{y(k+1)}$ reference its different version. Therefore, $S_{y(k+1)}$ can produce only the same results as $T_{x(k+1)}S_{y(k+1)}$.

(2) $S_{z(p+1)}$, ..., S_{zk} are not affected by $S_{y(k+1)}$.

Proof: We assume at least one of the statements in $S_{z(p+1)}$, ..., S_{zk} is affected by $S_{y(k+1)}$. Suppose S_{zh} is the closest statement to $S_{y(k+1)}$ among the statements that is affected by $S_{y(k+1)}$. That is, there is not any statement between S_{zh} and $S_{y(k+1)}$, or statements between S_{zh} and $S_{y(k+1)}$ should not be affected by $S_{y(k+1)}$. Because S_{zh} is affected, it must reference at least one item version that is installed by $S_{y(k+1)}$.

Suppose x_i is one of the item versions that are installed by $S_{y(k+1)}$ and referenced by S_{zh}. $S_{y(k+1)}$ and $T_{x(k+1)}S_{y(k+1)}$ produce the same results, so $T_{x(k+1)}S_{y(k+1)}$ also installs x_i. Suppose $T_{xh}S_{yh}$ is the corresponding statement of S_{zh} in the transaction execution sequence. Because $T_{xh}S_{yh}$ and S_{zh} are the same statement and thus use the same predicate, $T_{xh}S_{yh}$ must reference one of the versions of item x. Suppose the item version is x_j. x_j must be different from x_i as $T_{xh}S_{yh}$ is executed before $T_{x(k+1)}S_{y(k+1)}$, which is the last statement in transaction execution, and $T_{xh}S_{yh}$ cannot reference an item version that has not been installed yet. There are only two possible cases:

(a) $x_i \gg x_j$. $T_{xh}S_{yh}$ reference x_j while $T_{x(k+1)}S_{y(k+1)}$ installs x_i, and $x_i \gg x_j$, so $T_{x(k+1)}S_{y(k+1)}$ depends on $T_{xh}S_{yh}$. According to the topological sort, S_{zh} must be before $S_{y(k+1)}$. However, S_{zh} is after $S_{y(k+1)}$. Conflicts.

(b) $x_i \ll x_j$. Suppose x_j is installed by $T_{xj}S_{yj}$. Because $T_{xj}S_{yj}$ installs x_j and $T_{xh}S_{yh}$ reference x_j, $T_{xh}S_{yh}$ depends on $T_{xj}S_{yj}$. $T_{xj}S_{yj}$ installs x_j while $T_{x(k+1)}S_{y(k+1)}$ installs x_i, and $x_i \ll x_j$, so $T_{xj}S_{yj}$ depends on $T_{x(k+1)}S_{y(k+1)}$. So $T_{x(k+1)}S_{y(k+1)} \ll T_{xj}S_{yj} \ll T_{xh}S_{yh}$. According to topological sorting, $S_{y(k+1)}$ must be before S_{zj}, and S_{zj} must be before S_{zh}. Because S_{zh} is the closest statement that is affected by $S_{y(k+1)}$, and S_{zj} is before S_{zh}, S_{zj} is not affected by $S_{y(k+1)}$. So S_{zj} will also install x_j. Therefore, S_{zh} should use a version later than or equal to x_j. However, S_{zh} references version x_i that is older than x_j. Conflicts.

Combining (a) and (b), we can get that there is no statement in [$S_{z(p+1)}$, ..., S_{zk}] that is affected by $S_{y(k+1)}$. Therefore, $S_{z(p+1)}$, ..., S_{zk} should produce the same results as they produce in transaction execution.

Combining (1) and (2), we can get that [$T_{x1}S_{y1}$, $T_{x2}S_{y2}$, ..., $T_{xk}S_{yk}$, $T_{x(k+1)}S_{y(k+1)}$] and its topological sorting produce the same results. So for n = k + 1, the theorem still holds. Therefore, Theorem 1 is proved.