Fucci: Database Transaction Fuzzing via Random Conflict Construction and Multilevel Constraint Solving

Xiyue Gao
Xidian University
xygao@xidian.edu.cn

Zhuang Liu
Xidian University
zliu_01@stu.xidian.edu.cn

Yiran Shen
Xidian University
yrshen@stu.xidian.edu.cn

Hui Li*
Xidian University
hli@xidian.edu.cn

Yingfan Liu
Xidian University
liuyingfan@xidian.edu.cn

Hongjun Xiao
Xidian University
23031212454@stu.xidian.edu.cn

Yanguo Peng
Xidian University
ygpeng@xidian.edu.cn

Jiangtao Cui
Xidian University
cuijt@xidian.edu.cn

ABSTRACT

Ensuring the ACID properties of transactions is the fundamental functionality of transactional DBMSs. However, through our study on existing solutions on transaction management, we found that transaction implementations in some mainstream databases, such as MySQL, MariaDB and TiDB, may violate what they claim in their documentation, in the form of incorrect database state or query results. Since there is still a lack of efficient and comprehensive testing methods to detect bugs within transaction management implementation for off-the-shelf DBMSs at present, we propose Fucci, a fuzzing framework, to solve the problem. Given a target DBMS, Fucci improves the efficiency of detecting transaction bugs through three key components: Random Conflict Construction (RCC), Multilevel Constraint Solving (MCS), and Experience-driven Automatic Simplification (EAS). RCC addresses the issue of inadequate case validity by ensuring the presence of read-write or write-write conflicts between transactions. MCS enhances the accuracy and efficiency of the transaction oracle by employing an external multi-version control system to solve data visibility. EAS is ultimately adopted to improve the efficiency of simplification and the readability of the identified bug cases. All of the above strategies are tested on commercial databases such as MySQL, MariaDB and TiDB. Accordingly, 6 previously unknown transaction bugs and 14 known duplicate transaction bugs have been newly discovered, most of which have been officially acknowledged.

PVLDB Reference Format:
Xiyue Gao, Zhuang Liu, Yiran Shen, Hui Li, Yingfan Liu, Hongjun Xiao, Yanguo Peng, and Jiangtao Cui. Fucci: Database Transaction Fuzzing via Random Conflict Construction and Multilevel Constraint Solving. PVLDB, 18(6): 1879 - 1891, 2025.
doi:10.14778/3725688.3725713

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Reverie4u/Fucci.

*Hui Li is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725713

1 INTRODUCTION

Database Management Systems (DBMS), serving as the foundational infrastructure for managing and organizing data, have extensive applications in almost all software systems [1]. Ensuring their reliability and security is of paramount importance. A database transaction refers to a logical unit of work consisting of a series of operations, characterized by four properties: Atomicity, Consistency, Isolation and Durability (ACID) [28]. It is utilized to maintain the integrity of the database and has become an indispensable feature of modern DBMSs.

To effectively support ACID, DBMSs employ complex techniques. Undo/redo logs or protocols such as ARIES [38] ensure atomicity and durability. Lock-based concurrency control, multi-version concurrency control (MVCC) [33, 44] and optimistic concurrency control (OCC) [34, 47] are used to guarantee isolation. However, these technologies often exhibit design and implementation flaws, resulting in transaction management bugs [15, 16], which may manifest themselves as incorrect query results, blocking states and final database states. Therefore, conducting a thorough test for transaction management is crucial prior to the release of a DBMS.

Fuzzing, as a popular automated testing technique, detects software bugs by randomly generating and executing test cases [2, 4–6, 8, 29]. Compared to manual testing, symbolic execution [20] and model check [24], fuzzing offers lower cost and higher coverage. Currently, fuzzing has been extensively used to detect crashes, logic bugs, and performance bugs in DBMSs [36]. It works by randomly generating SQL statements and constructing appropriate test oracles [29] to verify the correctness of the execution results, thus detecting potential bugs in DBMSs.

Unfortunately, existing fuzzing methods for detecting transaction management bugs, such as DT2 [25] and Troc [26], suffer from low testing efficiency and accuracy. Specifically, DT2 [25] utilizes differential testing to verify transaction correctness, which can only validate the same syntax among different databases and fails to detect bugs shared by the tested and reference DBMSs. Although it is able to detect numerous compatibility issues between different databases, it can only identify a few transaction management bugs. Troc [26] considers confirming transaction management bugs by comparing the execution results under two equivalent scenarios: transactional and non-transactional modes. In non-transactional mode, the visible views of statements are first captured and dumped into temporary tables in the database. Then, the statements are executed on these temporary tables to simulate transaction execution.

However, its correctness relies on the accuracy of individual statement executions, which could lead to missed detections due to bugs in expression calculations within single statements. Moreover, existing methods employ entirely random methods to generate transaction test cases, often resulting in numerous invalid test cases and lower efficiency in vulnerability detection. More importantly, the test cases in the bug reports generated by DT2 and Troc require manual simplification to locate bugs, which greatly hinders both the reporting and the fixing of these bugs.

To effectively detect transaction management bugs in modern DBMSs, we introduce Fucci, a fuzzing tool based on Random Conflict Construction (RCC), Multilevel Constraint Solving (MCS) and Experience-driven Automatic Simplification (EAS). We start by randomly generating a database and transaction pairs. Then, we introduce random read-write conflicts or write-write conflicts between transaction pairs, providing a rich and effective selection space for transaction test cases. Next, we develop a constraint-solving oracle for testing the transaction functionality of DBMSs. It simulates transaction execution under different isolation levels using an external multi-version chain, an Abstract Syntax Tree (AST)-based statement solver and refined transaction analysis, and can obtain the ground truth of transaction execution. Finally, we present a bug case simplification model that maintains syntactic validity. This model prioritizes simplification layers and incorporates dynamic probability tables with the Epsilon-Greedy strategy to facilitate self-learning and decision-making for simplification operations.

To demonstrate the effectiveness and efficiency of Fucci, in line with experimental target DBMSs from previous research [26, 45], we conducted experiments in the pessimistic transaction mode across three widely-used DBMSs compatible with the MySQL protocol, i.e., MySQL [11], MariaDB [10], and TiDB [12]. Fucci is also applicable to other DBMSs like SQL Server, PostgreSQL and Oracle, but due to their different SQL dialects and isolation level definitions, extensive engineering adaptations are required, which will be a focus of our future efforts. Experiments show that Fucci identified six unique bugs in the transaction modules of the three tested DBMSs: one each in MySQL and TiDB, and four in MariaDB. These bugs have been reported to the developers and confirmed as new bugs. In particular, the MariaDB development team extensively discussed the cause of bug MDEV-27992 and proposed a preliminary solution, which includes introducing additional error codes and the 'innodb_snapshot_isolation' switch. One of the developers even commented 'Thank you for the great work! May I ask which tool you used to find this?'. This paper's contributions are as follows:

(1) We develop a multilevel constraint-solving oracle for detecting bugs in DBMS transaction management modules, effectively solving the false positive problem caused by relying on the correctness of single-statement executions.
(2) We introduce random conflicts and the hierarchical automatic simplification to enhance the effectiveness and readability of transaction test cases, thereby improving the efficiency of fuzzing.
(3) We implement these methods as Fucci and employed it to test three widely-used DBMSs, i.e., MySQL, MariaDB and TiDB. Fucci detected six unique transaction management bugs in these DBMSs.

2 PRELIMINARIES

2.1 Notations

The notations used in this paper are summarized in Table 1.

Table 1: Commonly Used Notations in This Paper

Notation	Definition
RU	Read Uncommitted
RC	Read Committed
RR	Repeatable Read
SER	Serializable
Tx	A transaction, which begins with 'BEGIN' and ends with 'COMMIT' and contains some statements
S	A statement in a transaction, which can be any SQL sentence supported by the target DBMS
t	A table in a database
p	A predicate in a SQL statement
$RS(S)$	The result set of the statement S
$IS(S)$	The insertion set of the INSERT statement S

2.2 Transaction Management

A transaction is a logical unit comprising a sequence of operations (e.g., read, write, insert) and should satisfy the ACID properties.

Transaction modes. There are two main modes: *optimistic* and *pessimistic*. Optimistic transactions skip row locks and validate conflicts at commit time, potentially requiring rollbacks. Pessimistic transactions acquire exclusive locks and block others until the locks are released. This work focuses on the pessimistic mode.

Isolation levels. Transactions operate under different isolation levels that balance consistency and concurrency. Higher levels offer stronger consistency but reduce concurrency. Different DBMSs may support various isolation levels. MySQL-compatible DBMSs typically support four levels: *RU*, *RC*, *RR*, and *SER*.

2.3 DBMS Fuzzing

Fuzzing is an efficient technique to identify issues in DBMSs by automatically generating and executing invalid, abnormal, or random test cases. It typically involves the following components:

Generator. Generator automatically produces test cases and serves as the first step of DBMS fuzzing. Test cases can be created using either generation-based [26, 41, 45] or mutation-based [19, 23, 35] methods. Mutation-based approaches rely on the quality of seed test cases and are more error-prone. We adopt the generation-based approach for better coverage and robustness.

Oracle. The oracle determines the expected outcomes of test cases. Existing oracles can be divided into four types: crash oracles [46, 49], differential oracles [25], metamorphic oracles [43] and constraint-solving oracles [41, 45]. Constraint-solving oracle, used in our work, generally relies on forward or backward solving to derive the ground truth of transaction execution.

Comparator. Comparator compares the execution results from the target DBMS with the expected results obtained from the oracle. If the results are inconsistent, it indicates a bug; otherwise, it confirms the system is functioning correctly.

Reducer. Reducer can automatically parse complex cases and perform simplification operations to obtain the simplified and human-readable format. However, existing reducers [23, 41] are limited to simplifying single SQL statements, not entire transactions.

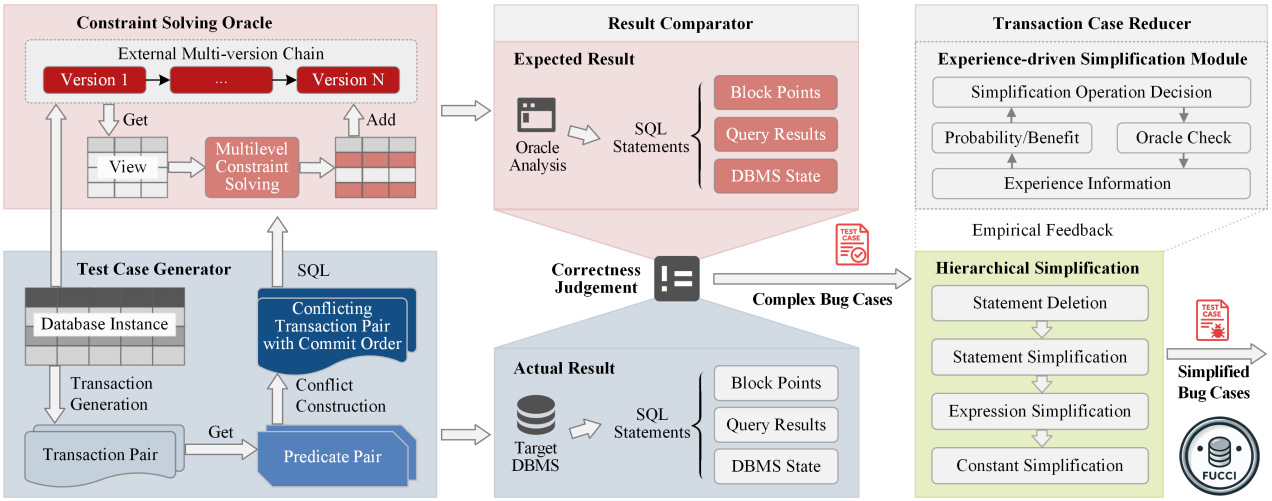

Figure 1: Overview of Fucci Framework

3 APPROACH

Overview. Figure 1 presents an overview of Fucci, divided into four modules. **Test case generator** employs a basic AST model mentioned in our previous review [27] to generate random databases and SQL statements. These statements are then combined with BEGIN, COMMIT or ROLLBACK to form transactions. We further enhance the effectiveness of these transaction cases through RCC, obtaining conflicting transaction pairs. Previous transaction oracles [26], based on mutation testing, often faced false positives due to reliance on the correctness of single-statement executions. Fucci's **constraint-solving oracle** is a fully external oracle that guarantees precise execution results by leveraging a multi-version chain, an AST-based statement solver and refined transaction analysis. While weak isolation levels might allow multiple correct outcomes due to indeterminate transaction commit orders, Fucci ensures unique results by fixing transaction commit order. **Result comparator** compares the results of the target DBMS and the oracle from three aspects: blocking condition, query result and final state of the database, to detect isolation bugs. The blocking condition ensures the blocking state of the oracle and actual execution align. Fuzzing methods often produce extremely complex test cases and therefore rely on manual simplification to enhance readability, which is inefficient [27]. In Fucci's **transaction case reducer**, a hierarchical simplification model simplifies complex transaction bug cases, while an experience-driven simplification module uses empirical feedback to guide the selection of simplification operations. They work together to generate most simplified bug reports. Next, we will elaborate on the core innovations of these modules.

3.1 Random Conflict Construction

Based on the definition of transaction anomalies [22], transactions will conflict on certain data items when dirty read and fuzzy read occur and also conflict within some predicate scope when phantom occurs. Additionally, analysis of known transaction bugs also shows they all involve read-write or write-write conflicts. To detect potential transaction bugs, it is necessary to construct test cases that contain the above conflicts to increase the probability of triggering anomalies. Table 2 shows all SQL statement types supported by Fucci, where 13 combinations can be used to construct conflicts. Notably, the combination of two common SELECT statements cannot trigger conflicts. Similarly, we do not consider the condition that two different INSERT statements add the same record because DBMSs will directly report an error or rollback the operation instead of of causing an transaction level conflict. However, SELECT (FOR SHARE/UPDATE) are two types of read statements that lock selected rows during queries. As a result, their combinations are considered viable candidates. To effectively construct conflicts, we introduce three strategies: *Fully Shared Filters (FSF)*, *Partially Shared Filters (PSF)* and *Conflict Tuple Containment (CTC)*.

Table 2: Statement combinations and conflict strategies

No	Statement A	Statement B	Strategy
1	SELECT (FOR SHARE)	SELECT (FOR SHARE)	
2	SELECT (FOR UPDATE)	SELECT (FOR UPDATE)	
3	UPDATE	UPDATE	
4	DELETE	DELETE	
5	SELECT	UPDATE	FSF
6	SELECT	DELETE	PSF
7	SELECT (FOR SHARE/UPDATE)	UPDATE	CTC
8	SELECT (FOR SHARE/UPDATE)	DELETE	
9	UPDATE	DELETE	
10	SELECT	INSERT	
11	SELECT (FOR SHARE/UPDATE)	INSERT	
12	UPDATE	INSERT	CTC
13	DELETE	INSERT	

FSF. Given that the SELECT, UPDATE and DELETE statements generated by the generator all involve predicates, it is easy to ensure they access the same items by keeping the predicates consistent. The left side of Figure 2 illustrates an example of *FSF*. Firstly, randomly select a statement from each of the two generated transactions, such as S_2 and S_7, where p_1 and p_2 respectively represent predicate expressions of the two statements. Then, change p_2 to p_1 to obtain a new statement $S_7^{'}$. Finally, since the predicate p_1 is fully shared by $S_7^{'}$ and S_2, it satisfies $RS(S_2) = RS(S_7^{'})$.

PSF. This strategy involves making the predicate of one statement contain that of another. In other words, the tuples filtered by one predicate become a subset of those filtered by the other. In

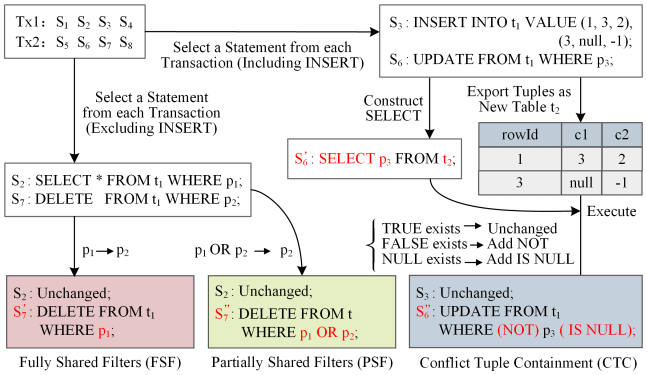

Figure 2: An Example of Random Conflict Strategies

Figure 2, the statements for *PSF* are the same as those for *FSF*. However, by altering p_2 to $(p_1; OR; p_2)$, a new statement S_7'' is created, satisfying $RS(S_2) \subseteq RS(S_7'')$.

CTC. Since INSERT statements have no predicates, we propose *CTC* to facilitate conflict construction of them. In Figure 2, S_3 is an INSERT statement, while S_6 is an UPDATE statement with the predicate p_3. We dump the tuples to be inserted from the INSERT into a new table t_2, which has the same table definition as the original table t_1. Then, construct a SELECT statement S_6' and execute it on table t_2 to verify whether the inserted rows satisfy the predicate p_3. If the result contains $TRUE/1$, it indicates a conflict between the inserted data and p_3. If the result contains $FALSE/0$ or $NULL$, a conflict can still be created by appending NOT or $IS\ NULL$ to p_3. Once the conflict is constructed, it satisfies $RS(S_6'') \cap IS(S_3) \neq \emptyset$. *CTC* can also be extended to non-INSERT scenarios (No.1-13 in Table 2). To achieve this, we first still randomly select some tuples from the test table into a new table, simulating the rows inserted by an INSERT. Then, for each statement randomly chosen from the two transactions, we construct two SELECT statements which are then executed on the new table to verify whether the selected tuples satisfy these predicates. Conflicts can be constructed by either adding NOT or $IS\ NULL$ or by not modifying them, using a similar strategy with INSERT scenarios.

Commit Order. While the above strategies generate conflicting statements, not all transaction commit orders trigger conflicts. We propose three filtering rules to exclude invalid orders: (1) The execution timelines of the two transactions must overlap; (2) The first transaction must not commit before the second statement in the combination is executed; (3) For consecutive BEGIN statements, only commit orders with different BEGIN sequences are redundant. Despite filtering, many orders may still remain. Exhaustively testing all of them is inefficient and unlikely to reveal new bugs. Therefore, we randomly sample k commit orders for testing. Empirical data indicates that $k = 10$ achieves highest detection efficiency with minimal cost. All experiments in this paper use this value.

Randomness in *RCC* is reflected in several aspects: (1) Random generation of initial transactions and databases; (2) Random selection of statements for conflict construction; (3) Conflict construction strategies can also be randomly chosen; (4) In CTC, when TRUE, FALSE and NULL coexist, whether to add NOT or IS NULL is also random. These random mechanisms allow RCC to maximize the test case search space within an effective range.

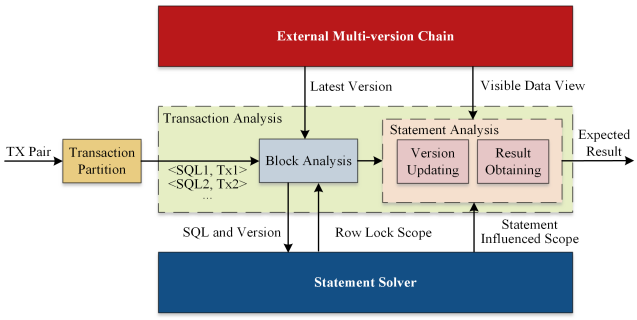

Figure 3: An Illustration of Multilevel Constraint Solving

3.2 Multilevel Constraint Solving

Constraint-solving oracle in Fucci constructs a fully external multi-version control system. It maintains a version chain for each tuple in test table and performs multilevel constraint solving on specified views to obtain the ground truth of transaction execution.

Figure 3 illustrates the process, including four key steps: transaction partition, block analysis, version updating and result obtaining. The latter three rely on a statement solver and a multi-version chain. Transaction partition involves dividing transactions into individual statements identified by transaction IDs. These IDs facilitate the analysis of data visibility of statements, as visibility varies from one transaction to another. Block analysis refers to obtaining the row lock of a single statement and checking for conflicts with existing locks. If a conflict exists, the current statement is blocked, and the process switches to a statement from the other transaction for execution. For unblocked UPDATE, DELETE and INSERT statements, version updating is performed, meaning new versions or deletion tags are inserted into the influenced scope of the solved statement to maintain the external multi-version chain. For unblocked SELECT statements, the visible data view of each statement is solved on the version chain based on the statement type and isolation level. Finally, through AST-based statement solving, we obtain the expected execution results, including the blocking condition of each statement, execution results and the final state of the database.

Statement Solver. In a multi-branch tree representation of an expression, leaf nodes typically represent constants or column names, while non-leaf nodes correspond to N-ary operators. Expression solving involves assigning a tuple to the column nodes of this tree and solving it from the bottom up to the root node. Algorithm 1 outlines this process, where the input is the root node *root* and a tuple list *tuples*, and the output is a result list *results*. For each tuple in *tuples*, the recursive function $SolveValue$ is invoked, and the results are added to *results*. If *root* is a leaf node, $SolveValue$ returns its constant value or retrieves the value from the tuple. Otherwise, it recursively evaluates child nodes and combines their results at the parent node. The final *results* contains values of $TRUE$, $FALSE$ or $NULL$ for each tuple. Using this approach, the statement solving process becomes straightforward: traverse *results* to locate rows where the expression evaluates to $TRUE$, thereby determining the influenced scope for SELECT, UPDATE, and DELETE statements.

Block Analysis. Locking mechanisms vary across different DBMSs. For example, MySQL supports gap locks [7], while TiDB does not. Table 3 summarizes the lock strategies used in MySQL,

Algorithm 1: Expression solving based on AST model

Input: Expression root node - *root*, table tuples - *tuples*
Output: Solved results - *results*

```
Function SolveValue(root, tuple):
    if root is a leaf node then
        if root is constant then
            return constant value;
        else
            return column value from tuple;
    else
        foreach child of root do
            subResult[i] ← SolveValue(child, tuple);
        res ← Combine all subResult to obtain the result of
          the parent expression;
        return res;
results ← ∅;
foreach tuple in tuples do
    result ← SolveValue(root, tuple);
    Add result to results;
return results
```

MariaDB and TiDB. MySQL and MariaDB share identical lock strategies, whereas TiDB supports only *RC* and *RR* isolation levels. In these DBMSs, regular SELECT statements are generally executed using snapshot reads: under *RU*, they read the latest data; under *RC*, they read from latest committed snapshot and under RR, they read from snapshot at the transaction begin. Locking is applied only under *SER* to ensure SELECT strict serializability. SELECT FOR SHARE and FOR UPDATE are two modes of current read, applying shared and exclusive locks, respectively, to the queried data. Both MySQL and MariaDB fully support these two reading modes, whereas TiDB does not support FOR SHARE by default. Although TiDB is compatible with LOCK IN SHARE MODE syntax (equivalent to FOR SHARE), it defaults to treating it as a regular, non-locking read. Additionally, under *RR* and *SER* in MySQL and MariaDB, most locking read and write operations include predicate locks to mitigate the effects of phantom reads.

The locking rules above clarify whether a statement will acquire a lock under different isolation levels. To analyze blocking, it is also necessary to determine the rows and index ranges involved when locking occurs. These ranges are closely related to the WHERE expression of the statement. Typically, the row lock range is obtained by solving the expression on the latest version of the test table. However, under *RC* of MySQL and MariaDB, semi-consistent reads [14] are used for UPDATE, and the WHERE condition is therefore evaluated based on the latest committed version.

Based on this information, the row lock of a statement can be modeled as a triplet comprising the lock type, row range and index range. The lock type is either shared or exclusive. The row range includes the tuples accessed by the statement, while the index range is defined as a pair consisting of the index name and corresponding values, which can be a primary key or other unique index. The left part of Figure 4 illustrates the lock analysis process for an UPDATE statement under the *RR* isolation level. First, the latest

Table 3: Lock strategies in different DBMSs. Lock types include shared locks (S LOCK) and exclusive locks (X LOCK), which can apply to rows (R), indexes (I), and predicates (P).

Type	MySQL & MariaDB				TiDB	
	RU	RC	RR	SER	RC	RR
SELECT	No LOCK (latest data)	No LOCK (latest committed snapshot)	No LOCK (transaction begin snapshot)	S LOCK (R, I, P)	No LOCK (latest committed snapshot)	No LOCK (transaction begin snapshot)
SELECT FOR SHARE	S LOCK (R, I)		S LOCK (R, I, P)		No LOCK (latest committed snapshot)	No LOCK (transaction begin snapshot)
SELECT FOR UPDATE	X LOCK (R, I)		X LOCK (R, I, P)		X LOCK (R, I)	
UPDATE	X LOCK (R, I)		X LOCK (R, I, P)		X LOCK (R, I)	
DELETE						
INSERT	X LOCK (R, I)				X LOCK (R, I)	

version $(null, -1), (2, 1), (null, -1)$ is obtained from the external multiversion chain. Then, the *Statement Solver* is used to determine the set of row numbers (2) that satisfy the conditions of the statement. According to the *RR* isolation level's locking strategy, the lock type is an exclusive lock. Based on the index metadata, the analysis identifies the locked index ranges as $< primary, (2, 1) >$ and $< index(c2), (1) >$, corresponding to the primary key index $(2, 1)$ and the unique index on column $c2$ with value (1), respectively.

For all statements after *Transaction Partition*, the row lock are analyzed sequentially and recorded. If the lock range of the current statement overlaps with that of previously accessed statements, the transaction containing the current statement is blocked until the other transaction commits or aborts (releasing all its locks). Under *RR* and *SER*, MySQL and MariaDB employ gap locks in different degrees, making it necessary to check for potential gap lock conflicts. Gap locks lock the gaps between tuples that satisfy predicate expressions to prevent phantom reads caused by insertion or deletion of tuples meeting the conditions. While determining precise gap lock ranges is computationally expensive, identifying conflicts between gap locks and row locks does not require exact ranges. Consider two statements, *A* and *B*, from different transactions. If the row lock analysis for statement *B* changes before and after the execution of statement *A* on the multi-version chain, it indicates a conflict between gap locks and row locks. The same holds true if the execution order of *A* and *B* is reversed.

External Multi-version Chain Maintenance. Considering that INSERT, UPDATE and DELETE statements all modify tuples in a table, we use an external multi-version chain to store the modification history of each tuple. To uniquely identify each tuple, a row number is maintained, with each corresponding to a linked list of historical versions. A version of a tuple is represented as a triple consisting of a data row, a transaction number and a deletion tag. The transaction number specifies which transaction added the version, and the deletion tag indicates whether the tuple has been deleted. For INSERT statements, a new tuple is generated with a new line number and its initial version is added to the version chain. Generally, determining the influenced scope of UPDATE and DELETE can follow the same approach as in block analysis. But the definition of the latest version differs slightly across isolation levels in MySQL and MariaDB. Under *RR*, unlike other levels, the

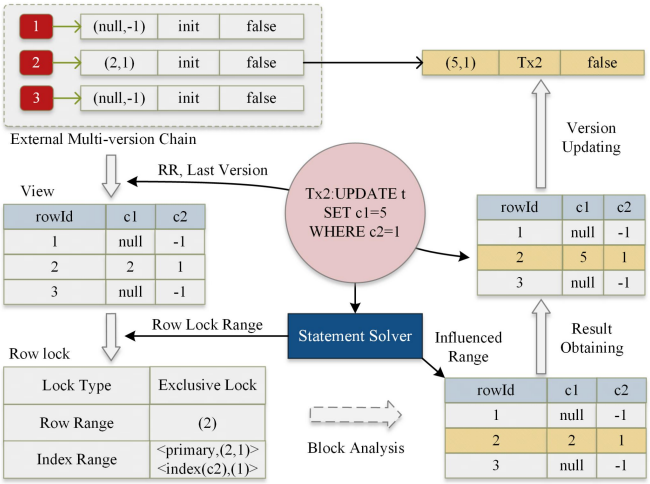

Figure 4: An Example of Multilevel Constraint Solving

latest version includes deleted rows. Once determined, the influenced range results leads to updates in the multi-version chain. For UPDATE statements, the updated tuple is added as a new version in the chain. For DELETE statements, the latest version is copied, its deletion tag is set to deleted, and it is added as a new version. Figure 4 also shows the process of maintaining the external multi-version chain. Initially, each tuple has a single version. The visible data view for *Tx2* is retrieved, and the expression $c2 = 1$ is evaluated to determine the scope, identified as (2). The visible view is then modified, and a new updated version is added to the corresponding row, updating the $c1$ column to 5.

Result Obtaining. Table 3 provides an explanation of the visibility rules for SELECT statements under the *RU*, *RC*, and *RR* isolation levels. In the *SER* isolation level, due to full transaction serialization, SELECT, FOR SHARE, and FOR UPDATE statements can only see versions committed by other transactions before the current transaction begins. However, under the *RU*, *RC*, and *RR* isolation levels, the snapshots for FOR SHARE and FOR UPDATE statements are always the latest committed snapshot. This is because these operations require acquiring shared or exclusive locks, which causes the current statement to be blocked until the conflicting write transaction is committed, ultimately allowing only committed versions to be read. Notably, our research differs from TROC in terms of data visibility. In particular, the difference between TROC and our work in data visibility is that the UPDATE, INSERT and DELETE statements in our work always obtain the latest version to construct the data view, while TROC has special rule definitions for them. Finally, the visible range is determined by evaluating the WHERE expression on these visible versions, leading to the expected results.

Limitation. MCS focuses on the dual-transaction scenario as most transaction bugs can be reproduced in it. However, there may indeed be a few bugs that need to be triggered in three or more concurrent transactions scenarios. If MCS is extended to the multi-transaction scenario, it will face uncertainty in oracle construction. For example, given three transactions $Tx1$, $Tx2$ and $Tx3$, if both $Tx1$ and $Tx2$ are blocked by $Tx3$, then the recovery order of $Tx1$ and $Tx2$ is uncertain after $Tx3$ commits, which will make oracle fail to obtain accurate results. A straightforward solution is to enumerate all possible recovery orders. If none match the observed execution, a transaction bug is detected. However, this approach is clearly inefficient, making it a key problem for future research.

Algorithm 2: Hierarchical simplification

Input: *text, maxReduceCount, categoryCount*
Output: *simplifiedText*

```
Tx1, Tx2, isolationLevels, submitOrder ← text
stmtList, astList ← Tx1, Tx2
testCase ← txParse(text)
foreach categoryCnt do
    foreach maxReduceCount do
        clonedTestCase ← testCaseClone(testCase)
        op[j] ← getType(reduceOpList[i])
        stmtIdx ← getRandomStmt(clonedTestCase)
        clonedTestCase ← performStmt(op[j], stmtIdx)
        if oracleCheck(clonedTestCase) == false then
            testCase ← clonedTestCase
simplifiedText ← textCase
```

3.3 Experience-driven Automatic Simplification

To enhance simplification efficiency and accuracy, we propose EAS, comprising a hierarchical simplification model and an experience-driven simplification module.

Hierarchical Simplification. We prioritize simplification layers as follows: statement deletion, statement simplification, expression simplification, and constant simplification. Operations within the same layer share equal priority. Statement deletion types match 3. Statement simplification includes removing expressions, updating columns, table definitions and projection columns. Expression simplification removes operators from WHERE clauses, while constant simplification targets complex constants in predicates. These AST-based operations ensure syntactic validity.

Algorithm 2 illustrates the hierarchical simplification process. Firstly, the input *text* is parsed, and up to *maxReduceCount* simplification operations are performed in the *categoryCnt-th* reduction layer. In each operation, clone the *testCase* and select simplification operations based on the decision algorithms. Then, randomly simplify a conditional statement from the *clonedTestCase*. If the simplified *clonedTestCase* still triggers a bug (*oracleCheck* returns false), it updates the current *testCase*. The final output $simplifiedText$ is the most simplified, readable test case.

Experience-driven Simplification. Although operations within the same layer have equal theoretical priority and can be randomly selected, their success rates and effects vary, making selection challenging. To address this, we propose two simplified sequential decision algorithms, *dynamic probability table* and *Epsilon-Greedy algorithm*, which optimize future decisions through feedback from simplification results and self-learning. The *dynamic probability table* algorithm maintains a probability table for each layer, recording the selection probability of each simplification operation. If the simplification type SELECT is chosen in the statement deletion layer, a SELECT statement from $Tx1$ or $Tx2$ is randomly deleted.

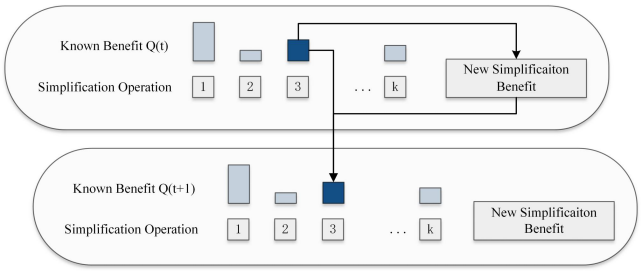

Figure 5: Updating process of known simpilificaiton benefit

The oracle then verifies if the bug is reproduced and updates the probabilities based on the simplification results.

The number of simplification operations is limited, as excessive simplification is time-consuming and may block the program. To maximize effectiveness within this limit, we model it as a multi-armed bandit problem and adopt the *Epsilon-Greedy algorithm*, selecting the highest-profit operation with a certain probability and randomly choosing the next with the rest.

$$\text{action}(t) = \begin{cases} \arg\max_{i=1...k}(Q_i(t)) & (1-\epsilon) \\ \text{random}(k) & (\epsilon) \end{cases} \tag{1}$$

$$Q_i(t+1) = Q_i(t) + \frac{g_i - Q_i(t)}{t+1} \tag{2}$$

The simplification module uses Eq. 1 to select the *t-th* simplification operation, where k is the number of candidate operations, $Q_i(t)$ represents the known profit of the *i-th* operation at time t, and ϵ is a fixed probability between 0 and 1. At each step, the operation with the highest profit is selected with probability $1-\epsilon$, while a random operation is chosen with probability ϵ. $Q_i(t+1)$ is updated based on the *i-th* operation's profit at time t and the simplification profit g_i (see Figure 5). The known profit $Q_i(t+1)$ is calculated as a standard average because the profit contribution of the *i-th* operation is consistent across iterations (Eq. 2). The simplified profit g_i is determined by oracle validation: if the simplified bug is reproducible, the profit is 1; otherwise, it is 0.

Adaptability. Fucci currently supports only MySQL-compatible DBMSs but has the potential to be extended to other DBMSs. For Fucci's generator, adapting to database dialects and adjusting for differences in data types, functions, and cross-database CRUD operations is essential, including updates to the AST model. Existing dialect conversion tools may assist in this process. Due to differences in isolation levels and locking strategies, adapting to Oracle introduces additional challenges, requiring clear definitions of lock types, granularity, and snapshot read rules to adjust Fucci's data view strategy. The statement solver also needs syntax adjustments, and the simplifier may need to optimize or expand its hierarchical strategies for better compatibility.

4 EVALUATION

To validate the superiority of Fucci, we conducted experiments across multiple DBMSs, including overall performance, oracle comparison, and conflict construction. Additionally, we analyzed the reasons behind the experimental results.

Table 4: New transaction bugs reported by Fucci

Issue	Isolation	Transaction Bug	Status	Severity	Affected Versions
MySQL #113228	RR	Yes	Verified	Serious	8.0, 8.2
MariaDB #32898	RR	Yes	Fixed	Critical	10.4-10.6, 10.11, 11.0-11.2
MariaDB #33802	RU, RC, RR, SER	Yes	Fixed	Critical	10.2-11.4
MariaDB #34106	RC	Yes	Verified	Critical	10.6, 10.11, 11.0-11.2, 11.4
MariaDB #34108	RU, RC, RR, SER	Yes	Verified	Critical	10.6
TiDB #48960	RR	Yes	Verified	Minor	7.1

4.1 Experimental Setup

We conducted experiments on MySQL 8.0.25, TiDB 7.1.2, and MariaDB 11.2.2 on a machine configured with a 16-core AMD Ryzen 7-5800H, running Ubuntu 20.04, and 16GB of RAM. The test case generator runs in a single-threaded environment with minimal system configuration requirements.

In the overall performance comparison experiment, we compared the bug detection efficiency of DT2, Troc, and Fucci. In the oracle comparison experiment, we further evaluated the oracle-solving capabilities of these fuzzers. DT2 and Troc were selected as baselines because other database fuzzing tools, such as SQLsmith [31], Squirrel [49], and NoREC [41], focus primarily on non-transactional modes. They generate SQL statements to detect bugs related to syntax, logic, memory, and performance optimization but are unable to identify transaction-related bugs involving isolation levels. Additionally, DT2 claims to be the first fuzzing tool that uses differential testing to detect transaction bugs [26], while Troc detects transaction bugs through transaction decoupling and mutation testing, making them ideal benchmarks for this study. In the conflict construction comparison experiment, we assess the effectiveness of FSF, PSF and CTC algorithms using a consistent oracle to validate the results. In these experiments, our SQL statements generation algorithm is the same for all cases.

4.2 Overall Evaluation

To evaluate the effectiveness of our fuzzing tool Fucci, we conducted approximately two weeks of testing on above-mentioned DBMSs. This effort resulted in the successful discovery of six new transaction bugs. Table 4 presents all the new transaction bugs detected in various databases. These six bugs lead to incorrect query results, with three causing phantom rows. One bug in MySQL is labeled serious, four in MariaDB as critical, and one in TiDB as minor. Currently, MariaDB#32898 and MariaDB#33802 are both fixed. They have been persistent issues since version 10.4, respectively. To fix MariaDB#32898, MariaDB introduced a new switch, 'innodb_snapshot_isolation', enabling a stricter snapshot isolation level compared to the default repeatable read. To fix MariaDB#33802, the MariaDB developers improve the error code 'db_record_changed' and the function of cursor restoration. We will provide more details about these bugs in Section 5.

Additionally, we compared Fucci with the state-of-the-art fuzzers DT2 [25] and Troc [26] for detecting transaction bugs. We compared the number of unique and duplicate bugs within the same

Table 5: Comparision of different fuzzers

Fuzzer	MySQL Bugs			MariaDB Bugs			TiDB Bugs			Total	Avg Latency
	Unique	Duplicate	Total	Unique	Duplicate	Total	Unique	Duplicate	Total		
DT2 [25]	1	4	5	1	1	2	0	0	0	7	10.3 h
Troc [26]	2	2	4	2	4	6	1	2	3	13	5.5 h
Fucci	3	3	6	2	10	12	2	2	4	22	3.3 h

time frame and evaluated the average time to find a bug. The experimental results are presented in Table 5. The difference between a unique and a duplicate bug is whether the cause of the bug is the same as the cause of these identified bugs. Meanwhile, whether the bugs found in our work are unique or duplicate is defined by the database development team after we submit these bugs, so the evaluation of these found bugs is objective and authoritative. Both MariaDB and MySQL utilize InnoDB as their storage engine, potentially leading to vulnerabilities stemming from historical legacy issues associated with InnoDB. In contrast, TiDB's storage engines, TiKV and TiFlash, are built from scratch, mitigating such concerns. In general, our proposed fuzzing tool, Fucci, detected the highest number of total bugs and the shortest average detection latency across all three DBMSs. After 24 hours of testing, Fucci found 22 bugs, which is 3.1x more than DT2 and 1.7x more than Troc. On average, Fucci discovered a bug every 3.3 hours, while DT2 took 10.3 hours and Troc took 5.5 hours.

Specifically, on MySQL, Fucci identified the highest number of unique and total bugs, and the second-highest number of duplicate bugs, while DT2 identified the highest number of duplicate bugs and the second-highest number of total bugs. The effectiveness of DT2 may be due to the divergence of development teams for MySQL and MariaDB since an early version. Since then, these teams have added distinct features or accidentally introduced new bugs, which DT2 can catch. On the other hand, on MariaDB and TiDB, thanks to random conflicts construction and multilevel constraint solving, Fucci detected the most unique bugs and duplicate bugs, while DT2 performed the worst. The significant differences in MVCC and locking mechanisms between the test and reference databases lead to numerous false positives and affect the effectiveness of DT2. For instance, during testing on TiDB, DT2 generated over 100 bug reports, yet none of them turned out to be transaction bugs.

We also performed cross-validation to check whether transaction bugs reported by DT2 and Troc could be confirmed by Fucci, and vice versa. Fucci's multilevel constraint-solving oracle successfully verified all 4 transaction bugs reported by DT2 and all 10 reported by Troc. However, DT2 only detected 1 out of 4 bugs reported by Fucci, as the remaining 3 were common bugs shared across the test DBMSs, beyond DT2's testing capabilities.

4.3 Comparison of test oracles

We conducted separate experiments on the test oracle, comparing our method with related works on differential oracle and metamorphic testing to validate the effectiveness of the multilevel constraint-solving oracle. In particular, the differential oracle adopts a pairwise comparison method. For instance, when testing MySQL, MariaDB serves as the reference DBMS. The implementation of the differential oracle is consistent with DT2, while the implementation of the metamorphic oracle aligns with Troc. Our experiments use the same generator to generate statements and transactions, meaning that the set of test cases solved by all oracles is identical.

The experimental results in Figure 6 demonstrate that each oracle achieves an accuracy of over 98%, indicating the effectiveness of both related works and our oracle. Moreover, there is a notable disparity in the cases solved per minute and correctly solved cases per minute for the metamorphic oracle across different databases. It performs best on MariaDB and least effectively on TiDB, primarily due to performance variations among the tested databases. We conducted experiments on a single-node TiDB, thus obtaining lower efficiency in statement and transaction execution compared to standalone MySQL and MariaDB instances. Additionally, metamorphic testing requires executing test cases twice on the target database, once in transaction mode and once in non-transaction mode, significantly magnifying performance disparities.

From Figure 6 (a), (d), and (g), it is obvious that on MySQL, our multilevel constraint-solving oracle outperforms all the baselines in the aspects of accuracy (over 99.99%), cases solved per minute, and correctly solved cases per minute, respectively. As shown in Figure 6 (a), our oracle exhibits higher accuracy compared to DT2. It is because of the significant differences in the implementation of transaction management between MySQL and MariaDB, which may result in false positives for the differential oracle. Our oracle does not show a significant improvement in accuracy compared to Troc, as Troc already achieves an accuracy of over 99.95%, and further improvement is challenging. However, our oracle exhibits a 52% increase in the cases solved per minute and correctly solved cases per minute compared to Troc and a 97% increase compared to DT2. The primary reason is that our multilevel constraint-solving oracle operates independently of database execution, thus avoiding additional database connection overheads, leading to significantly higher solving efficiency for small-scale datasets.

From Figure 6 (b), (e), and (h), it is evident that on MariaDB, our oracle achieves the highest accuracy (100%), as well as the highest number of cases solved per minute and correctly solved cases per minute. It indicates that our method performs better overall on MariaDB than other methods. Figure 6 (b) indicates a minimal accuracy gap among the three oracles on MariaDB. It is because the differences in SQL syntax and locking mechanisms between MariaDB and TiDB are relatively small, causing DT2's accuracy to approach that of the other two oracles.

From Figure 6 (c), (f), and (i), it's evident that our oracle continues to outperform others on TiDB. Unlike MySQL and MariaDB, the number of cases solved per minute and correctly solved cases per minute for metamorphic testing is lowest on TiDB. Troc requires executing test cases twice on TiDB, while DT2 executes them once on TiDB and MySQL, and our oracle executes them only once on TiDB. The performance differences between TiDB and the other two contribute to the experimental results in Figure 6 (f) and (i).

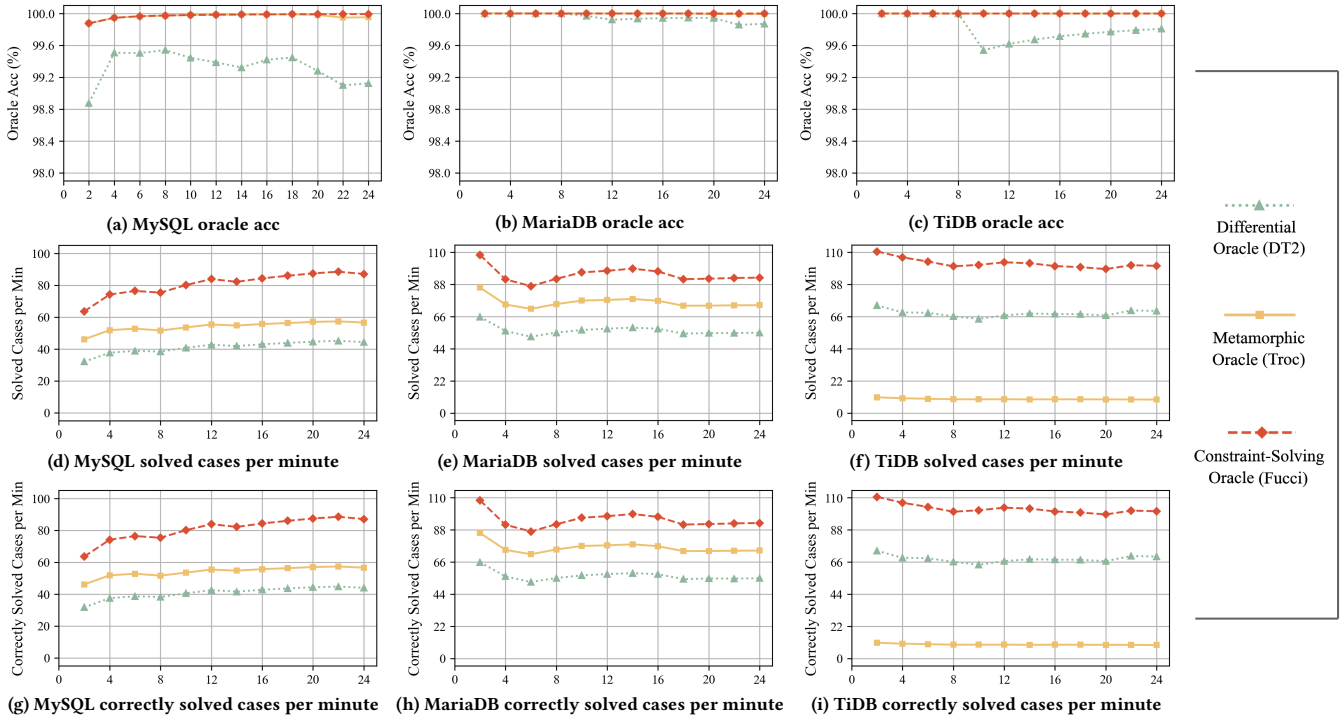

Figure 6: Comparison between different oracles. This figure shows the oracle accuracy, cases solved per minute, and correctly solved cases per minute over time. We run each oracle for 24 hours in MySQL, MariaDB, and TiDB.

In conclusion, the multilevel constraint-solving oracle outperforms DT2 and Troc across the three DBMSs. One reason is that its constraint-solving oracle operates independently of the database execution's correctness, effectively avoiding missed detections in metamorphic testing caused by erroneous execution statement results. It also mitigates false positives in differential testing arising from inconsistent transaction concurrency control mechanisms across DBMSs. Another reason is that the constraint solver operates faster than database execution for small-scale datasets, leading to faster solving speed compared to DT2 and Troc.

4.4 Evaluation of random conflict construction

We conducted a separate evaluation of the algorithms for constructing random conflict to demonstrate their effectiveness. The experiment utilized the same statement generator and oracle, with evaluation metrics including total bugs, total cases, and case quality. Among them, case quality is defined as the ratio of total bugs to total cases, reflecting the contribution of individual cases to bug detection. The experimental results are shown in Figure 7.

The results show that all conflict construction algorithms can increase the number of detected bugs within the same timeframe. Among them, the CTC algorithm detects the most bugs across all test DBMSs and exhibits the highest case quality. It is attributed to the CTC algorithm's ability to construct partial conflicts between transactions without increasing transaction complexity, thus enhancing bug detection efficiency. Similarly, the PSF algorithm can achieve partial conflicts but is less effective than CTC. It is because transactions generated by the PSF algorithm tend to have longer execution times, resulting in the fewest test cases executed within 24 hours, approximately 89.3% compared to no conflicts.

The performance of algorithms on MySQL and TiDB is similar. Although FSF detects more bugs than PSF, it also executes more cases, resulting in similar case quality. With the total cases being similar among None, FSF, and CTC, we compared their total bugs and concluded that partial conflicts between transactions are more effective than full conflicts, which are more effective than no conflicts. On MariaDB, the PSF algorithm outperforms the FSF algorithm. We conducted an in-depth study of all newly discovered bugs on MariaDB and found that most bugs can only be triggered by partial conflicts, further validating our conclusion.

4.5 Comparison of Simplification Algorithms

We conducted separate experiments on MySQL, MariaDB and TiDB with different simplification algorithms. To demonstrate the effectiveness and applicability of experience-driven simplification algorithms, we compared them with the random simplification algorithm under the same database. The experiments utilized the same transaction bug detection components and hierarchical simplification model, with reduction rate and validity are selected as the evaluation indexes. Among them, reduction rate is defined as the proportion of the number of characters that have been simplified to the number of original case's characters, and reduction validity

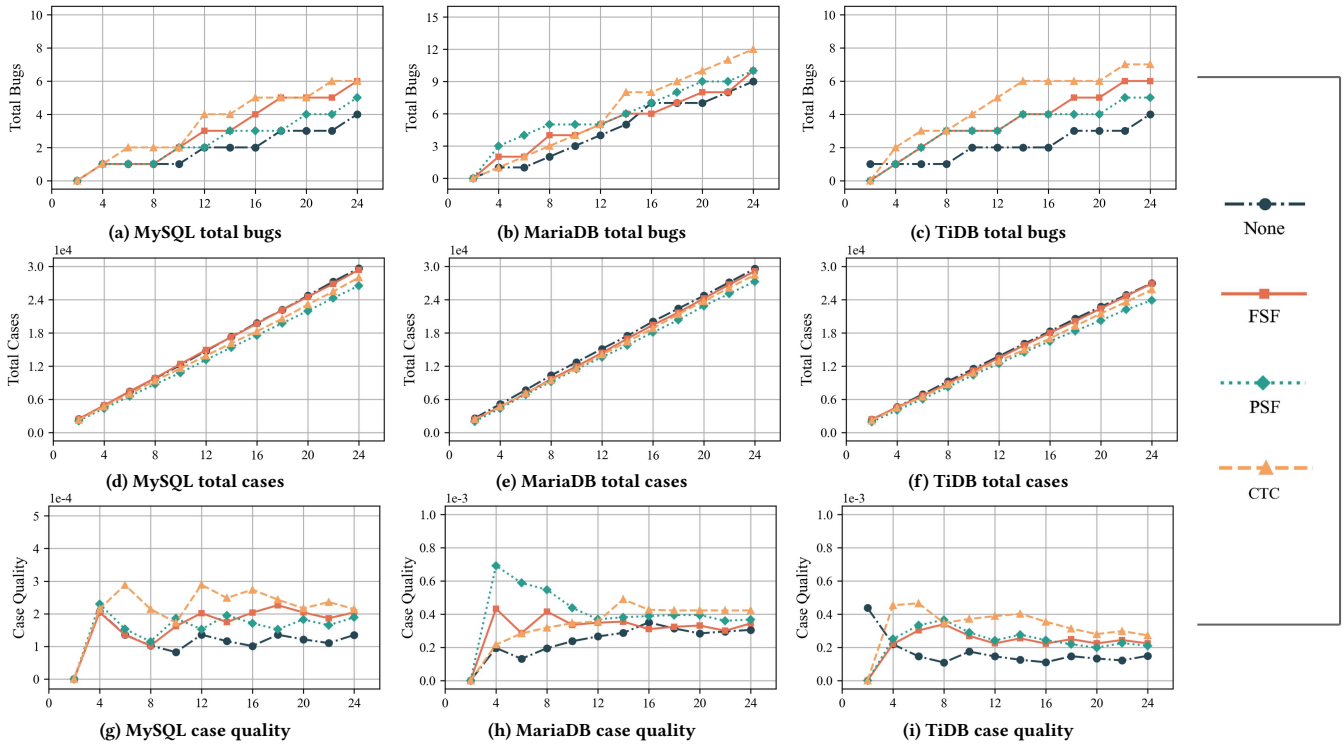

Figure 7: Comparison between different conflict construction algorithms. This figure shows the number of total bugs and total cases as well as case quality. We test each algorithm for 24 hours in MySQL, MariaDB, and TiDB.

refers to the ratio of successful simplification times to the total simplification times. These two reflect the simplification degree and efficiency of cases respectively.

Figure 8 reveals that initially, the random simplification algorithm achieves the highest simplification rates, overall reduction validity and first-level reduction validity across three test databases. The inferior performance of the probability table and Epsilon-Greedy algorithms in the early stages is due to lack of experiential information. As the testing progresses, the performance of the Epsilon-Greedy and the probability table algorithms gradually surpasses that of the random simplification algorithm, with the Epsilon-Greedy algorithm ultimately performing the best.

Figure 8 (a), (b) and (c) indicate that when the number of simplification is certain, the final simplification rate of each algorithm under different databases stabilizes at 29.5%, 29% and 28%. At the same time, both the probability table and the Epsilon-Greedy algorithm are higher than the random simplification algorithm, which indicates that the hierarchical simplification model proposed in this paper has the ability to simplify cases to the same complexity in any database, and the experience-driven information algorithm can effectively improve its simplification ability.

Figure 8 (d), (e) and (f) show that the effectiveness of empirical algorithms in different databases improves over time, reaching 43% and 39%, respectively. In contrast, the effectiveness of the random simplification algorithm decreases from 44% to 37%, highlighting the superior efficiency and applicability of experience-driven information algorithms in enhancing simplification.

Lastly, Figure 8 (g), (h) and (i) show that the reduction validity of the first level, specifically the statement deletion layer, surpasses the random approach with the guidance of the experience-driven simplification, stabilizing at values around 75%, 74%. This confirms the effectiveness of the experience-driven information simplification. In addition, regardless of which database is running, the reduction validity of the first level is nearly 32% higher than the overall one, highlighting the significant impact of the statement deletion layer in improving case simplification efficiency.

5 CASE STUDY

5.1 Phantom rows caused by UPDATE

Code Snippet 1 shows the bug of MySQL#113228 [16] and MariaDB#32898 [15], it can be reproduced on *RR* level. When an UPDATE statement modifies the primary key, phantom rows may appear during transaction execution and disappear after commitment.

In Code Snippet 1, lines 1-3 are the table creation statements, and lines 4-11 are SQL statements arranged in the order of transaction execution. The comments before the statements represent the transaction numbers, with the initial database belonging to the *init* session, and *Tx1* and *Tx2* denote two transactions under

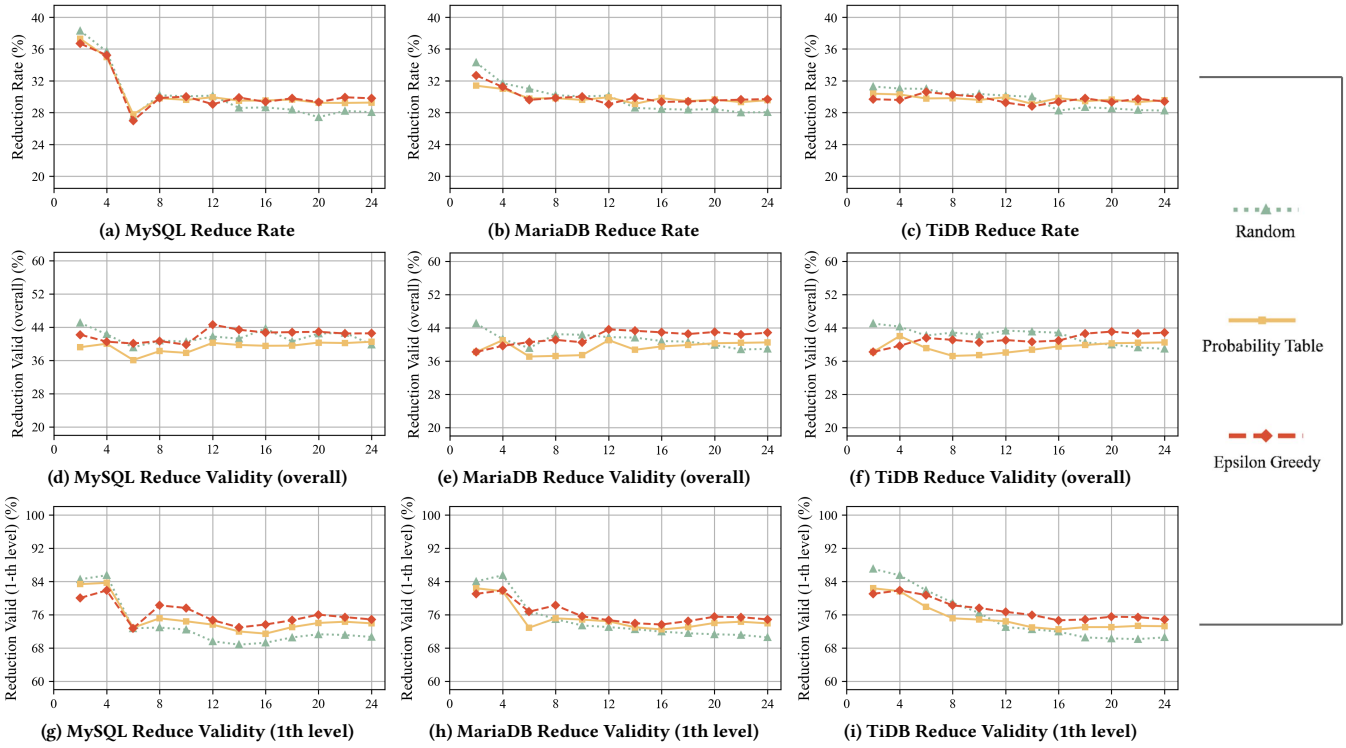

Figure 8: Comparison of Simplification Decision Algorithms. This figure shows the reduction rate and validity in MySQL, MariaDB and TiDB. We test each algorithm for 24 hours.

different sessions. First, *Tx2* queries the data from table t, returning $[(1,1),(2,2)]$ (line 6). Next, *Tx1* applies an exclusive lock to the row $(2,2)$ that satisfies the UPDATE predicate and creates a new version $(3,2)$ at primary key position 3 (line 7), releasing the lock upon commit (line 8). Then, *Tx2*'s UPDATE statement reads the latest versions of all data items, acquires locks, and generates new versions $(1,3)$ and $(3,3)$ (line 9). The query in line 10 of *Tx2* is a snapshot query, which can see the snapshot of *Tx2*'s own UPDATE. Therefore, the expected result is $[(1,3),(3,3)]$. However, the actual result is $[(1,3),(2,2),(3,3)]$, where $(2,2)$ is a shadow row that disappears after *Tx2* commits.

```
1  /*init*/ CREATE TABLE t(a INT PRIMARY KEY, b INT);
2  /*init*/ INSERT INTO t VALUES (1, 1);
3  /*init*/ INSERT INTO t VALUES (2, 2);
4  /*Tx1*/ BEGIN;
5  /*Tx2*/ BEGIN;
6  /*Tx2*/ SELECT * FROM t;-- [(1, 1), (2, 2)]
7  /*Tx1*/ UPDATE t SET a=3 WHERE b = 2;
8  /*Tx1*/ COMMIT;
9  /*Tx2*/ UPDATE t SET b=3;
10 /*Tx2*/ SELECT * FROM t;-- actual: [(1, 3), (2, 2),
       (3, 3)], expected: [(1, 3), (3, 3)]
11 /*Tx2*/ COMMIT; -- [(1, 3), (3, 3)]
```

Code Snippet 1: phantom rows caused by UPDATE

The cause of this bug lies in *Tx2*'s UPDATE statement (line 9), which locks the record $(2,2)$ that is invisible in *Tx2*'s snapshot, making it visible to subsequent statements in *Tx2*. The MariaDB community developers have fixed this bug by reporting an error 'db_record_changed' when 'innodb_snapshot_isolation' is on.

5.2 Weird read view after ROLLBACK

Code Snippet 2 shows the issue of bug MariaDB#33802 [17], it can be reproduced on isolation level *SER*. When a SELECT statement projects the primary key and UNIQUE columns before ROLLBACK, the same projection rows appear.

In Code Snippet 2, lines 1-2 are table creation statements, and lines 3-8 are SQL statements executed according to the transaction execution order. First, *Tx1* inserts row $[(1, null)]$ (line 5). Secondly, *Tx2* projects the primary key a and unique column b in table t (line 6). Then, before *Tx2* COMMIT (line 8), *Tx1* ROLLBACK (line 7). Consequently, the query result of *Tx2* changes to [(1, null), (1, null)].

```
1  /* init */ CREATE TABLE t(a INT PRIMARY KEY, b INT
       UNIQUE);
2  /* init */ INSERT INTO t(a) VALUES (1);
3  /* Tx1 */ BEGIN;
4  /* Tx2 */ BEGIN;
5  /* Tx1 */ INSERT INTO t(a) VALUES (2);
6  /* Tx2 */ SELECT a, b FROM t;-- actual: [(1, null),
       (1, null)], expected: [(1, null)]
7  /* Tx1 */ ROLLBACK;
8  /* Tx2 */ COMMIT;
```

Code Snippet 2: update shadow row bug

One reason for this bug is that the fix for MariaDB#32898 is not comprehensive. The error code 'db_record_changed' does not cover all exceptional cases. Another reason is that cursor restoration does not account for the possibility of multiple NULL records in a unique index. The MariaDB community developers have addressed this bug from both of these aspects.

5.3 Inconsistent behaviors of UPDATE

Code Snippet 3 demonstrates a transaction bug MariaDB#34108 [13] which can be reproduced on isolation level *RC*. When an UPDATE statement of $Tx2$ is executed between two UPDATE statements of $Tx1$, the query result after COMMIT of $Tx1$ and $Tx2$ is incorrect.

In Code Snippet 3, lines 1-2 are table creation statements, and lines 3-9 are statements executed in the committed order. Line 10 is an extra query of table t. First, $Tx1$ updates column b in table t (line 5), followed by $Tx2$ attempting to update column b (line 6). However, since $Tx1$'s UPDATE statement locks the tuples, $Tx2$'s UPDATE statement is blocked. Next, $Tx1$ sets column a to 1 (line 7). After $Tx1$ commits, the exclusive locks are released, and the current committed version becomes $[(1,3),(1,3)]$. Subsequently, $Tx2$'s UPDATE statement proceeds, updating all values in column b to 2 and then committing. Therefore, the expected snapshot query result should be $[(1,2),(1,2)]$, but the actual result is $[(1,3),(1,2)]$.

```
1  /* init */ CREATE TABLE t(a INT, b INT);
2  /* init */ INSERT INTO t VALUES (null, 1), (1, 1);
3  /* Tx1 */ BEGIN;
4  /* Tx2 */ BEGIN;
5  /* Tx1 */ UPDATE t SET b=3;
6  /* Tx2 */ UPDATE t SET b=2 WHERE a is not null;  --
       blocked
7  /* Tx1 */ UPDATE t SET a=1;
8  /* Tx1 */ COMMIT; -- statement of line 6 recovered
9  /* Tx2 */ COMMIT;
10 /* Tx1 */ SELECT * FROM t; -- actual: [(1, 3), (1 ,2)
       ], expected: [(1, 2), (1, 2)]
```

Code Snippet 3: inconsistent update behaviors bug

The bug likely stems from an implementation flaw where the UPDATE operation fails to lock records modified by uncommitted transactions during WHERE clause scans, opting instead for a semi-consistent, non-locking read.

6 RELATED WORK

In this section, we provide a concise overview of current progress in DBMS fuzzing. For an in-depth exploration, refer to our previous review [27]. Current research focuses on test case generation and oracle construction. Test case generation includes generation-based and mutation-based approaches. Oracle construction includes differential testing, mutation testing and constraint-solving testing.

Generator type and strategy. Generation-based methods are categorized into AST-based and Alloy-based models [30]. RAGS [42] adjusts AST model generation using a static configuration file, while APOLLO [32] enhances statement validity with a dynamic probability table. TQS [48] generates unique connection patterns by randomly traversing schema graphs. Alloy-based models [18] enable database fuzz tests with higher semantic correctness.

Mutation-based methods. SQL Server [21] uses an AST-based mutation method to generate test cases by adding, modifying, and deleting projection lists and clauses. However, strict type constraints and complex operations make AST mutation as challenging as modifying SQL statements directly. Squirrel [49] simplifies this by converting the AST into an Intermediate Representation (IR) and performing semantically correct mutations on the IR.

Differential Oracle. Differential oracles detect logical bugs or performance issues by comparing execution results across DBMSs or versions. RAGS [42] identifies bugs by comparing the execution results of the same SQL statements on different commercial databases. SQLSmith [31] and Go-randgen [9], popular open-source tools, also use differential oracles for results verification. DT2 [25] detects compatibility and transaction bugs by comparing transaction blocking, query results, and database states across databases. APOLLO [32] uses differential testing across database versions to detect performance regressions.

Metamorphic Oracle. Metamorphic oracles construct equivalent statements by transforming the original statement to ensure consistent execution results [3, 9, 21, 25, 42]. NoREC [40] rewrites statements to bypass database optimization, detecting incorrect optimizations. DQE [43] converts SELECT statements into equivalent UPDATE and DELETE statements. TLP [41] ensures predicate evaluations always fall within TRUE, FALSE, or NULL, enabling queries to be decomposed into three partitioned queries, also applicable for detecting transaction isolation bugs. Troc [26] tests two equivalent modes: transaction mode, submitting multiple transactions directly, and non-transaction mode, splitting transactions into individual statements with transaction identifiers.

Constraint-Solving Oracle. Not all queries can obtain expected results through constructing equivalent queries, so some fuzzing methods generally use constraint-solving oracles to generate the desired results. These methods [18, 37, 41, 45] infer the logical truth of query outputs through forward or backward solving. Forward solving evaluates each tuple based on predicate constraints using an external solver (e.g., SAT solver) to obtain the basic truth values of statement execution results [18, 37]. In this process, a logical solver can also be used instead of a SAT solver. TQS [48] accelerates this process by converting join predicates into logical operations. Backward solving [39] involves randomly selecting some tuples as the basis truth values and uses a SAT solver to work backward and obtain statements whose execution results include these tuples.

7 CONCLUSION

In this paper, we propose a novel, efficient and universal DBMS transaction fuzzing framework, Fucci. To tackle issues of inadequate case validity, low oracle accuracy and inefficient bug simplification, Fucci employs three novel techniques, RCC, MCS and EAS to generate valid transaction test cases with their expected results and obtain the most readable bug cases. We conducted experiments on MySQL, MariaDB, and TiDB, and successfully found 6 previously unknown transaction bugs and 14 known transaction bugs. We plan to further expand Fucci's engineering applications to adapt to more DBMSs. It can serve as an important tool for DBMS development.

ACKNOWLEDGMENTS

This work supported by National Natural Science Foundation of China under Grant No. 62302370, 62272369, 62372352, 62172314.

REFERENCES

[1] 2011. Data science revealed: A data-driven glimpse into the burgeoning new field. https://www.ndm.net/datawarehouse/pdf/EMC-Data_Science_Study_White_Paper.pdf. Last accessed September 1, 2024.

[2] 2015. *AFL: American fuzzy lop.* http://lcamtuf.coredump.cx/afl/ Last accessed September 1, 2024.

[3] 2015. *SQLsmith.* https://github.com/anse1/sqlsmith Last accessed September 1, 2024.

[4] 2016. *Honggfuzz.* https://google.github.io/honggfuzz/ Last accessed September 1, 2024.

[5] 2016. *OSS-Fuzz: Continuous fuzzing for open source software.* https://github.com/google/oss-fuzz Last accessed September 1, 2024.

[6] 2017. *LibFuzzer - A library for coverage-guided fuzz testing.* http://llvm.org/docs/LibFuzzer.html Last accessed September 1, 2024.

[7] 2018. *Mysql InnoDB Locking.* https://dev.mysql.com/doc/refman/8.0/en/innodb-locking.html Last accessed September 1, 2024.

[8] 2019. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl. Last accessed September 1, 2024.

[9] 2023. *go-randgen.* https://github.com/pingcap/go-randgen Last accessed September 1, 2024.

[10] 2023. *MariaDB.* https://mariadb.org/ Last accessed September 1, 2024.

[11] 2023. *MySQL.* https://github.com/mysql/mysql-server Last accessed September 1, 2024.

[12] 2023. *TiDB.* https://github.com/pingcap/tidb Last accessed September 1, 2024.

[13] 2024. Inappropriate semi-consistent read in RC. https://jira.mariadb.org/browse/MDEV-34108. Last accessed September 1, 2024.

[14] 2024. *Innodb transaction isolation levels.* https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html Last accessed September 1, 2024.

[15] 2024. Phantom rows caused by UPDATE of PRIMARY KEY. https://jira.mariadb.org/browse/MDEV-32898. Last accessed September 1, 2024.

[16] 2024. Phantom rows caused by update statements which changes value of the primary key. https://bugs.mysql.com/bug.php?id=113228. Last accessed September 1, 2024.

[17] 2024. Weird read view after ROLLBACK of other transactions. https://jira.mariadb.org/browse/MDEV-33802. Last accessed September 1, 2024.

[18] Shadi Abdul Khalek and Sarfraz Khurshid. 2010. Automated SQL query generation for systematic testing of database engines. In *International Conference on Automated Software Engineering*. ACM, 329–332.

[19] Jinsheng Ba and Manuel Rigger. 2023. Testing database engines via query plan guidance. In *International Conference on Software Engineering*. IEEE, 2060–2071.

[20] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *Comput. Surveys* 51, 3 (2018), 1–39.

[21] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. 2007. A genetic approach for random testing of database systems. In *International Conference on Very Large Data Bases*. VLDB Endowment, 1243–1251.

[22] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24 (1995), 1–10.

[23] Xinyue Chen, Chenglong Wang, and Alvin Cheung. 2020. Testing query execution engines with mutations. In *Workshop on Testing Database Systems*. ACM, 1–5.

[24] Edmund M Clarke. 1997. Model checking. In *Foundations of Software Technology and Theoretical Computer Science*. Springer, 54–56.

[25] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. 2022. Differentially testing database transactions for fun and profit. In *International Conference on Automated Software Engineering*. ACM, 1–12.

[26] Wensheng Dou, Ziyu Cui, Qianwang Dai, Jiansen Song, Dong Wang, Yu Gao, Wei Wang, Jun Wei, Lei Chen, Hanmo Wang, et al. 2023. Detecting isolation bugs via transaction oracle construction. In *International Conference on Software Engineering*. IEEE.

[27] Xiyue Gao, Zhuang Liu, Jiangtao Cui, Hui Li, Hui Zhang, Kewei Wei, and Kankan Zhao. 2023. A Comprehensive Survey on Database Management System Fuzzing: Techniques, Taxonomy and Experimental Comparison. *arXiv preprint arXiv:2311.06728* (2023).

[28] Jim Gray and Andreas Reuter. 1992. *Transaction processing: concepts and techniques*. Elsevier.

[29] William E Howden. 1978. Theoretical and empirical studies of program testing. *Transactions on Software Engineering* SE-4, 4 (1978), 293–298.

[30] Daniel Jackson. 2002. Alloy: A lightweight object modelling notation. *Transactions on Software Engineering and Methodology* 11, 2 (2002), 256–290.

[31] Matt Jibson. 2019. *SQLsmith: Randomized sql testing in cockroachdb.* https://www.cockroachlabs.com/blog/sqlsmith-randomized-sql-testing/ Last accessed September 1, 2024.

[32] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Very Large Data Bases Endowment* 13, 1 (2019), 57–70.

[33] Jongbin Kim, Jaeseon Yu, Jaechan Ahn, Sooyong Kang, and Hyungsoo Jung. 2022. Diva: Making MVCC Systems HTAP-Friendly. In *International Conference on Management of Data*. ACM, 49–64.

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

[35] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-oriented DBMS fuzzing. In *International Conference on Data Engineering*. IEEE, 668–681.

[36] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4309–4326. https://www.usenix.org/conference/usenixsecurity22/presentation/liang Last accessed September 1, 2024.

[37] Kaiming Mi, Chunxi Zhang, Weining Qian, and Rong Zhang. 2021. Artemis: An automatic test suite generator for large scale OLAP database. In *Benchmarking, Measuring, and Optimizing: Third BenchCouncil International Symposium*. Springer, 74–89.

[38] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems* 17, 1 (1992), 94–162.

[39] Manuel Rigger. 2019. *Cast of string with newlines to signed/unsigned returns unexpected result.* https://bugs.mysql.com/bug.php?id=96294 Last accessed September 1, 2024.

[40] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM.

[41] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 667–682.

[42] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *International Conference on Very Large Data Bases*. VLDB Endowment, 618–622.

[43] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing database systems via differential query execution. In *International Conference on Software Engineering*. IEEE, 2072–2084.

[44] Xiaohui Song and Jane W-S Liu. 1990. Performance of multiversion concurrency control algorithms in maintaining temporal consistency. In *Proceedings Fourteenth Annual International Computer Software and Applications Conference*. IEEE Computer Society, 132–133.

[45] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting logic bugs of join optimizations in DBMS. *International Conference on Management of Data* 1, 1 (2023), 1–26.

[46] Shihao Wen, Peng Jia, Pin Yang, and Chi Hu. 2023. Squill: Testing DBMS with correctness feedback and accurate instantiation. *Applied Sciences* 13, 4 (2023), 2519.

[47] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. Tictoc: Time traveling optimistic concurrency control. In *International Conference on Management of Data*. ACM, 1629–1642.

[48] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *USENIX Security Symposium*. USENIX Association, 2307–2324.

[49] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing database management systems with language validity and coverage feedback. In *Conference on Computer and Communications Security*. ACM, 955–970.