Detecting Schema-Related Logic Bugs in Relational DBMSs via Equivalent Database Construction

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

Wensheng Dou*†‡
Institute of Software at CAS, China
wsdou@otcaix.iscas.ac.cn

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

Yu Gao*
Institute of Software at CAS, China
gaoyu15@otcaix.iscas.ac.cn

Ziyu Cui*
Institute of Software at CAS, China
cuiziyu20@otcaix.iscas.ac.cn

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

ABSTRACT

Relational Database Management Systems (DBMSs) provide flexible DDL (Data Definition Language) statements that enable the creation, modification, and deletion of database schemas. In addition to database schemas, relational DBMSs typically manage various schema-related information internally, e.g., schema changes, tablespace allocation, and block-level data layout. However, incorrect implementations related to schema-related information maintenance and utilization can introduce schema-related logic bugs. These bugs can cause DQL (Data Query Language) statements to return incorrect query results and DML (Data Manipulation Language) statements to create incorrect database states. Existing approaches mainly focus on detecting logic bugs in DQL statements, but are ineffective in detecting schema-related logic bugs.

In this paper, we propose a novel and general testing approach, `DDLCheck`, to effectively detect schema-related logic bugs in relational DBMSs. We first generate a complex DDL sequence seq_{gen} that consists of various types of DDL statements, and then synthesize a rather simple DDL sequence seq_{syn}, which utilizes `CREATE` statements to create the same database schema as seq_{gen}. Executing the same SQL statements on the two databases created by seq_{gen} and seq_{syn} should yield the same execution results. Any discrepancy between their execution results indicates a schema-related logic bug. To improve the testing efficiency of `DDLCheck`, we further design a DDL-sequence-oriented testing optimization strategy, which can help `DDLCheck` explore diverse schema-related information and detect schema-related logic bugs quickly. We implement and evaluate `DDLCheck` on six widely-used relational DBMSs. We have detected 34 bugs in these DBMSs, of which 29 bugs have been confirmed as previously unknown bugs and 9 bugs have been fixed.

PVLDB Reference Format:
Jiansen Song, Wensheng Dou, Yingying Zheng, Yu Gao, Ziyu Cui, Wei Wang, and Jun Wei. Detecting Schema-Related Logic Bugs in Relational DBMSs via Equivalent Database Construction. PVLDB, 18(7): 2281 - 2294, 2025.
doi:10.14778/3734839.3734861

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

Proceedings of the VLDB Endowment, Vol. 18, No. 7 ISSN 2150-8097.
doi:10.14778/3734839.3734861

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://figshare.com/s/e222bd56c4925375ac34.

1 INTRODUCTION

Relational Database Management Systems (DBMSs) are the most commonly used DBMSs [2], offering efficient data storage, manipulation, and retrieval for many data-intensive applications, e.g., e-commerce and social applications [6]. These applications often change their database schemas to keep up with the changing requirements [23]. We refer to the process of changing database schemas as database schema evolution in our paper.

Relational DBMSs provide a variety of DDL (Data Definition Language) statements, e.g., `CREATE TABLE`, `ALTER TABLE`, and `DROP TABLE`, which enable DBMS users to flexibly manipulate database schemas. Database schemas usually consist of various database objects, including tables, views, columns, indexes, constraints, etc. For example, `ALTER TABLE` statements facilitate significant modifications to existing tables, enabling users to delete columns, change data types, etc.

In addition to user-visible database schemas, relational DBMSs internally manage various schema-related information for storing database schemas, managing database schema changes, managing physical storage information, and optimizing DBMS execution. For example, MySQL stores the structure of database schemas into the system table `INFORMATION_SCHEMA`. PostgreSQL manages physical storage information, e.g., tablespace allocation and block-level data layout. When executing DDL statements to manipulate database schemas, the database schemas themselves and their associated schema-related information will be changed accordingly.

Due to the complexity of database schemas and schema-related DBMS optimizations [29, 36], it becomes a challenging task for relational DBMSs to correctly maintain and utilize schema-related information. Incorrect implementations related to the maintenance and utilization of schema-related information can introduce schema-related logic bugs, *schemaBug*s for short. These logic bugs can

```
1. -- Database db1
2. CREATE TABLE t1 (c1 INT, c2 BLOB) ROW_FORMAT=REDUNDANT;
3. ALTER TABLE t1 DROP c2;
4. INSERT INTO t1 (c1) VALUES (0); - Throw an error: Row size too large 🐞
5. -- db1's state: t1:{} ✘

6. -- Database db2
7. CREATE TABLE t1 (c1 INT) ROW_FORMAT=REDUNDANT;
8. INSERT INTO t1 (c1) VALUES (0); -- No errors
9. -- db2's state: t1:{0} ✔
```

Listing 1: MDEV#35122. The INSERT statement fails due to the dropped column c2.

silently lead to incorrect query results for DQL (Data Query Language) statements or incorrect database states for DML (Data Manipulation Language) statements.

Listing 1 shows a real-world *schemaBug* MDEV#35122 detected by our approach in the widely-used DBMS MariaDB. This *schemaBug* affects INSERT statements on tables with the option *ROW_FORMAT =REDUNDANT* if they contain previously dropped BLOB columns. Specifically, if a table t1 is created with a BLOB column c2, and that BLOB column is subsequently dropped (Lines 2–3), MariaDB does not immediately remove c2 but marks it as a hidden column. When a user tries to insert data into other columns, e.g., c1 of table t1, MariaDB implicitly attempts to insert a NULL value for the hidden BLOB column, resulting in throwing an error "Row size too large" (Line 4). We reported this bug to MariaDB developers, who identified it as a new bug with *Critical* level and fixed it.

Existing testing approaches for relational DBMSs mainly focus on detecting logic bugs in the SELECT statements [15, 24, 31, 42, 43, 47, 48, 50, 51, 54]. Differential testing [50] feeds the same SELECT statement into multiple DBMSs and compares their returned query results. However, differential testing fails to test SQL features specific to individual DBMSs. Metamorphic testing [15, 31, 43, 47, 48] detects logic bugs in individual DBMSs by constructing equivalent SELECT statements and observing differences among their outputs. However, the above proposed metamorphic relations are not suitable for other SQL statements, since DDL statements and some DML statements generally do not have WHERE clauses like SELECT statements. Thus, existing approaches cannot be effectively applied to detect *schemaBug*s.

In relational DBMSs, different DDL sequences can create equivalent databases, which share the same user-visible database schema. DBMSs should return the same execution results for the same SQL statements executed on the equivalent databases. However, different DDL sequences adopt different execution logic to create the equivalent databases, which may result in the creation of different schema-related information, ultimately leading to inconsistent execution results of subsequent SQL statements. For example, databases db_1 and db_2 created by different DDL sequences (Lines 2–3 and Line 7) should store the same data after executing the same INSERT statement (Line 4 and Line 8) in Listing 1. However, MariaDB stores different data in db_1 and db_2, which indicates a *schemaBug*.

Inspired by the above key observations, we propose a novel approach DDLCheck to effectively detect *schemaBug*s in relational DBMSs with regard to database schema evolution. Specifically, we first generate a complex DDL sequence seq_{gen} that consists of various types of DDL statements, e.g., CREATE TABLE, ALTER TABLE, and DROP TABLE. In such way, we model a complex database schema evolution process. We then synthesize a simple DDL sequence seq_{syn}, which utilizes CREATE TABLE and CREATE VIEW statements, by contrast, creating the same database schema as seq_{gen} in a straightforward process. Next, we execute the DDL sequences seq_{gen} and seq_{syn} to create two equivalent databases db_{gen} and db_{syn}, respectively. We execute the same generated DML and DQL statements on these two databases and expect that they return the same execution results, i.e., storing the same data for DML statements and returning the same query results for DQL statements. If the same SQL statement executed on databases db_{gen} and db_{syn} returns inconsistent execution results, DDLCheck reports a *schemaBug*.

Different DDL sequences may produce very similar schema-related information, which can reduce the testing efficiency of DDLCheck. For example, DDL sequences *CREATE TABLE t0 (c1 INT); RENAME TABLE t0 TO t1* and *CREATE TABLE t2 (c1 INT); RENAME TABLE t2 TO t3* can produce two different database schemas. However, they share similar schema-related information except the final table names. To solve this problem, we propose a DDL-sequence-oriented testing optimization strategy to avoid testing the same or similar schema-related information. We analyze the table structure change history made by DDL sequences, and filter out similar DDL sequences, allowing DDLCheck to focus on diverse DDL sequences.

To evaluate the effectiveness of DDLCheck, we implement and evaluate DDLCheck on six widely-used relational DBMSs, i.e., MySQL [7], PostgreSQL [8], SQLite [10], MariaDB [5], CockroachDB [1], and TiDB [12]. In total, we have detected 34 bugs, of which 29 bugs have been confirmed as new bugs and 9 bugs have been fixed by DBMS developers. Moreover, our experimental results demonstrate that our proposed DDL-sequence-oriented testing optimization strategy can help DDLCheck test more unique DDL sequences and detect *schemaBug*s more quickly. We further compare DDLCheck with three state-of-the-art relational DBMS testing approaches, i.e., NoREC [47], DQP [15], and Radar [52], with respect to their bug detection capabilities. The comparison result shows that none of *schemaBug*s in DDL and DML statements can be detected by these approaches. The above experimental results show that DDLCheck is effective in detecting *schemaBug*s in relational DBMSs.

In summary, we make the following contributions.

- We propose the first DBMS testing approach DDLCheck with regard to database schema evolution. We detect schema-related logic bugs in relational DBMSs by constructing equivalent databases using different DDL sequences.
- We design a DDL-sequence-oriented testing optimization strategy to improve the testing efficiency of DDLCheck, thus testing diverse DDL sequences and detecting schema-related logic bugs quickly.
- We implement and evaluate DDLCheck on six widely-used relational DBMSs. DDLCheck has detected 34 bugs, of which 29 bugs have been confirmed as previously unknown bugs.

2 PRELIMINARIES

We first introduce relational DBMSs and our target DBMSs (Section 2.1). Then, we explain SQL statements in our target DBMSs (Section 2.2). Finally, we present schema-related information in our target DBMSs (Section 2.3).

Table 1: Target relational DBMSs

DBMS	DB-Engines Ranking	GitHub Stars	Type
MySQL	2	10.8K	Traditional
PostgreSQL	4	16.1K	Traditional
SQLite	10	7.1k	Embedded
MariaDB	15	5.6K	Traditional
CockroachDB	64	30.1K	NewSQL
TiDB	77	37.2K	NewSQL

2.1 Relational Database Management Systems

Relational Database Management Systems (DBMSs) are used to create, manage, and interact with relational databases. In a relational DBMS, data is organized into structured tables (relations) with predefined columns and rows, allowing for efficient storage, retrieval, and manipulation of data [19].

Table 1 shows our target DBMSs, including MySQL [7], PostgreSQL [8], SQLite [10], MariaDB [5], CockroachDB [53], and TiDB [34]. We select our target relational DBMSs by using the following criteria. First, the DBMS is open-source and provides convenient bug tracking systems, allowing us to actively interact with developers. Second, the DBMS should have been thoroughly tested by existing approaches [15, 47, 48, 51, 52]. Third, the DBMS can cover different types of DBMSs, e.g., traditional DBMSs and NewSQL DBSMs.

Among the 7 relational DBMSs in the top-10 systems on the DB-Engines Ranking [2], we select MySQL, PostgreSQL, and SQLite as our target DBMSs, since they are open-source, and we can easily access their communities and report bugs. We do not select the remaining four relational DBMSs (i.e., Oracle, Microsoft SQL Server, Snowflake, and IBM Db2) because they are commercial and closed-source DBMSs, and it is challenging for us to report and investigate bugs. We select MariaDB, CockroachDB, and TiDB, because they have been thoroughly tested by existing approaches [31, 38, 43, 44, 47, 48, 51, 52, 58, 63]. Moreover, our target DBMSs cover different types of DBMSs. Specifically, MySQL, PostgreSQL, and MariaDB are traditional DBMSs, TiDB and CockroachDB are NewSQL DBMSs, and SQLite is an embedded DBMS.

2.2 Structured Query Language

Relational DBMSs provide the powerful query language SQL (Structured Query Language) [18], enabling users to perform various operations including defining database schemas, inserting and updating database records, and retrieving data from the database. SQL statements are typically categorized into three main types based on their functionalities.

DDL (Data Definition Language) statements are used to define and modify database schemas. DDL statements include `CREATE TABLE`, `ALTER TABLE`, and `DROP TABLE` statements, which allow users to create new database objects, modify existing database objects, and remove existing database objects. For example, we can use `CREATE TABLE` statements to create new tables, `ALTER TABLE` statements to modify existing tables, and `DROP TABLE` statements to drop existing tables.

DML (Data Manipulation Language) statements are used to store and manipulate the data stored in a database. For example, we can use `INSERT` statements to add new data into a table, `UPDATE` statements to modify existing data in a table, and `DELETE` statements to remove data from a table.

DQL (Data Query Language) statements are used to retrieve data from the database. The most prominent DQL statements are `SELECT` statements, allowing users to filter, sort, and aggregate data from one or more tables based on specific conditions. `SELECT` statements contain lots of syntax features, e.g., `WHERE` clauses, `ORDER BY` clauses, aggregate functions, join clauses, and `GROUP BY` clauses.

2.3 Schema-Related Information

Relational DBMSs use database schemas to describe the data structure of their managed databases. A database schema consist of various database objects, including tables, views, columns, indexes, etc. To effectively store and manage these database objects, relational DBMSs support various schema-related information.

Relational DBMSs utilize schema-related information to store the structure of database schemas, accounting for the relationships and dependencies among various database objects. Different DBMSs use different methods to store schema structures. For example, MySQL stores database schemas into the system table `INFORMATION_SCHEMA`, while PostgreSQL maintains a comprehensive catalog of system tables, e.g., `pg_index`, `pg_attribute`, and `pg_constraint`. Specifically, PostgreSQL uses table `pg_index` to store index information, including the indexed tables, the indexed columns, and index-specific attributes (e.g., B-tree, hash, and spatial). Relational DBMSs also utilize schema-related information to manage database schema changes. For example, PostgreSQL records DDL statements when enabling `log_statement=` 'ddl'. Aside from the above information, relational DBMSs also store physical storage information about database schemas, e.g., file storage, tablespace allocation, and block-level data layout. For example, PostgreSQL organizes tables, views, columns, indexes, and associated data structures as files within the file system, typically under the `pg_data` directory.

Users can manipulate database schemas through various DDL statements. This not only modifies user-visible database schemas but also implicitly changes schema-related information. For example, when executing `ALTER TABLE t1 DROP COLUMN c1` statement on table `t1` with the option *ROW_FORMAT=REDUNDANT*, MariaDB uses the general log to record the database schema changes [37]. However, column `c1` is only hidden without physically rebuilding table `t1`. MariaDB still maintains schema-related information about `c1`. These schema-related information impact the execution of subsequent SQL statements. For example, MariaDB implicitly stores `NULL` values into the hidden column `c1` for `INSERT` statements.

The complexity of maintaining and utilizing schema-related information in relational DBMSs exposes them to several correctness threats. First, incorrect schema-related information maintenance can cause incorrect database schemas. For example, in MariaDB, `ALTER TABLE` statements forget to update associated `FOREIGN KEY` constraints, leading to incorrect database schemas [3]. Second, incorrect schema-related information usage can cause incorrect database states or incorrect query results. For example, in MariaDB, `REPLACE` statements omit newly added `UNIQUE` constraints, resulting in storing incorrect data [4].

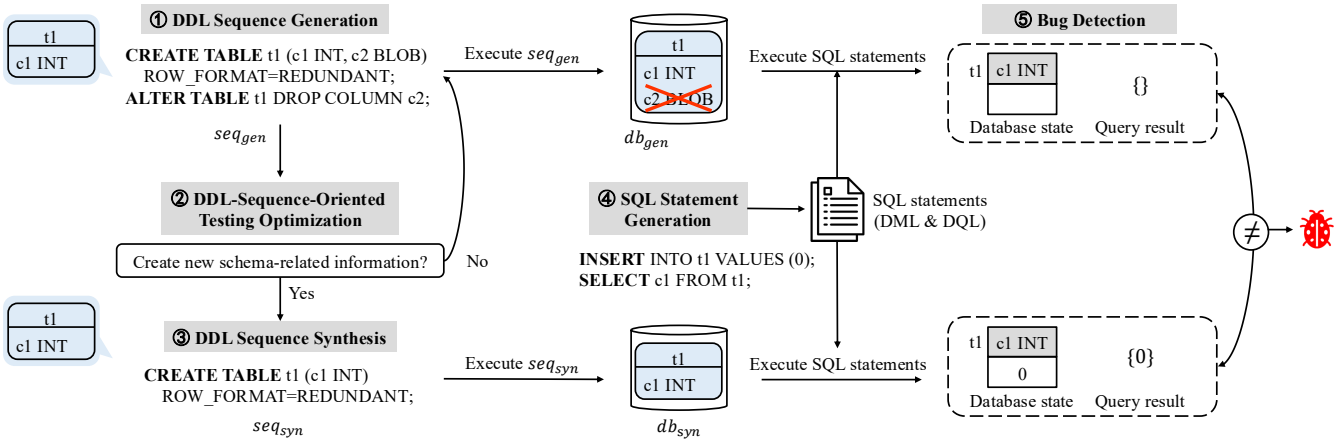

Figure 1: The architecture of DDLCheck.

3 APPROACH

We propose DDLCheck to effectively detect *schemaBugs* in relational DBMSs. The core idea of DDLCheck is that we can construct two different DDL sequences that create databases with the same database schema but different schema-related information, e.g., different file storage and different data layout. These databases should return the same execution results for the same SQL statements. Any discrepancy between their returned execution results indicates a potential *schemaBug* in the target DBMS. In this paper, we refer to databases that have the same database schema as equivalent databases.

3.1 DDLCheck's Architecture

Figure 1 shows the architecture of DDLCheck, which consists of five components. The DDL sequence generation component (①) randomly generates a complex sequence of DDL statements seq_{gen} consisting of various types of DDL statements (e.g., `CREATE TABLE` and `ALTER TABLE`) to create a database db_{gen}. The DDL sequence synthesis component (③) generates a simple sequence of DDL statements seq_{syn}, which uses `CREATE` statements (e.g., `CREATE TABLE`) to create a database db_{syn} with the same database schema as db_{gen}. After that, the SQL generation component (④) randomly generates a sequence of SQL statements (e.g., DML statements and DQL statements) to manipulate the databases db_{gen} and db_{syn}, where we expect to obtain the same query results and database states. The bug detection component (⑤) executes the generated SQL statements against databases db_{gen} and db_{syn}, then checks whether the target DBMS returns the same query results for the same DQL statements and maintains the same database states for the same DML statements. If the DBMS returns different execution results or database states, a *schemaBug* is reported.

Our randomly generated DDL sequences may produce the same or similar schema-related information, which can reduce the testing efficiency of DDLCheck. To address this problem, the DDL-sequence-oriented testing optimization component (②) filters out DDL sequences that create the same or similar schema-related information as previous generated DDL sequences. We repeat the above testing process until exhausting a fixed time budget.

$$
\begin{aligned}
genSeq &:=< createTable >< ddlStmt+ > \\
ddlStmt &:= createStmt \mid alterStmt \mid dropStmt \\
createStmt &:= createTable \mid createIndex \mid createView \\
alterStmt &:= alterTable \mid alterView \\
dropStmt &:= dropTable \mid dropIndex \mid dropView
\end{aligned}
$$

Figure 2: A formal description for generated DDL sequences.

3.2 DDL Sequence Generation

Enumerating all the possible database schemas allows for a more comprehensive test of DBMSs, but the large test space can detract from the effectiveness of DBMS testing. For example, MySQL supports 43 data types for a column. If we create two tables with three columns in a database, and consider two simple constraints (e.g., `NOT NULL` and `UNIQUE`) for each column, then we need to generate $(43 \times 2 \times 2)^{3\times2} \approx 2.59 \times 10^{13}$ database schemas in MySQL. It is almost impossible to test all these database schemas in a limited time budget (e.g., a month). Thus, random statement generation can be an effective approach to explore more database schemas within a limited time budget (e.g., 24 hours).

Randomly generating statements is a commonly-used and effective approach in DBMS testing [31, 47–49, 51, 52]. It can help to explore more database schemas within the limited time budget. Therefore, we utilize random statement generation to create valid DDL sequences. However, a completely randomized approach would hinder test space exploration because of the stateful nature of DBMSs [38]. For example, we cannot modify a table before it has been created.

To generate valid DDL sequences, we first build a formal representation to describe the legal transformations among DDL statements, as shown in Figure 2. A DDL sequence begins with a `CREATE TABLE` statement (i.e., *createTable*), followed by some other DDL statements (i.e., *ddlStmt*). A DDL statement can be a `CREATE` statement *createStmt* (e.g., *createTable*), an `ALTER` statement *alterStmt* (e.g., *alterTable*), or a `DROP` statement *dropStmt* (e.g., *dropTable*).

Algorithm 1: Dynamic DDL Sequence Generation

Input: The maximum length of DDL sequences *maxLength*
Output: A DDL sequence *seq*

```
1  schema ← ∅ // Start with an empty database schema
2  seq ← [] // Initialize an empty DDL sequence
3  ddlStmt ← generateCreateTable()
4  seq ← seq ∪ [ddlStmt]
5  schema ← updateSchema(schema, ddlStmt)
6  currentLength ← random(1, maxLength)
7  while |seq| < currentLength do
8      ddlStmt ← generateDDLstatement(schema)
9      seq ← seq ∪ [ddlStmt]
10     schema ← updateSchema(schema, ddlStmt)
11 return seq
```

Based on the formal representation of DDL sequence, we randomly generate a valid DDL sequence consisting of various DDL statement types (e.g., *createTable*, *alterView*, and *dropIndex*). Algorithm 1 illustrates the generation of DDL sequence. Starting with an empty database schema *schema*, we first generate a `CREATE TABLE` statement *ddlStmt* (Lines 1–4), ensuring other DDL statements can operate on existing tables. We then execute statement *ddlStmt* to update the database schema *schema* (Line 5), which will guide the generation of subsequent DDL statements. We continue to generate additional DDL statements until the total number of DDL statements reaches a randomly generated length *currentLength* that is between 1 and *maxLength* (Lines 6–7). In each iteration, we randomly select a certain type of DDL statement (e.g., `CREATE INDEX`) and then generate a concrete DDL statement for it. When a DDL statement is generated, the database schema *schema* is updated accordingly (Line 10). Finally, we can generate a complex DDL sequence with various types of DDL statements.

In our experiments, we set the maximum length of generated DDL sequences *maxLength* to 10. We generate concrete DDL statements following the way described in Section 3.6. When generating a single DDL statement, `DDLCheck` supports a wide variety of DDL statements with diverse syntax features. We utilize a grammar-based approach to generate DDL statements. DDL options are compiled into DBMS-specific grammar files. If a DDL statement contains some options, we randomly generate these options with suitable values. For example, in MySQL, `ALTER TABLE` statements have the `ALGORITHM` option with the acceptable range of values including `INSTANT`, `INPLACE`, and `COPY`. We support 26, 23, 11, 24, 19, and 26 types of DDL statements across MySQL, PostgreSQL, SQLite, MariaDB, CockroachDB, and TiDB, respectively.

3.3 DDL Sequence Synthesis

After generating a complex DDL sequence seq_{gen}, we first execute seq_{gen} to obtain the generated database schema *schema* from the target DBMS. We use *SELECT * FROM INFORMATION_SCHEMA* to obtain the generated database schema in MySQL, MariaDB, CockroachDB, TiDB. In SQLite, we obtain database schemas from `SQLite_SCHEMA`. In PostgreSQL, we separately obtain the database objects in *schema* from the corresponding system table. For example,

Algorithm 2: CREATE TABLE Statement Synthesis

```
1  stmt ← 'CREATE TABLE'
2  stmt ← stmt + ' ' + getTableName() // e.g., t1
3  stmt ← stmt + '('
4  while hasNextColumn() do
5      stmt ← stmt + ' ' + getColumnName() // e.g., c1
6      stmt ← stmt + ' ' + getDataType() // e.g., int
7      if getColumnKey() = UNI then
8          stmt ← stmt + ' ' + 'UNIQUE'
9      if getColumnKey() = PRI then
10         stmt ← stmt + ' ' + 'PRIMARY KEY'
11     // handle other column attributes
12     ...
13 while hasNextReferenceTable() do
14     stmt ← stmt + ' ' + 'FOREIGN KEY (' +
         getColumnName() + ') REFERENCES' // e.g., c1
15     stmt ← stmt + ' ' + getReferTable() + '(' +
         getReferColumn() + ')' // e.g., t0(c2)
16 // handle other table attributes
17 ...
18 return stmt
```

we obtain the definition of indexes from the `pg_index` table. Next, we attempt to synthesize a simple DDL sequence seq_{syn} to create the same database schema as *schema*. In seq_{syn}, all database objects are created and defined using the `CREATE TABLE` and `CREATE VIEW` statements, without using other DDL statements such as `ALTER` or `DROP` statements.

Algorithm 2 illustrates how to construct a concrete *createTable* statement. Specifically, we initialize a *createTable* statement *stmt* with the `CREATE TABLE` syntax terms and append a table name (Lines 1–3). Next, we iteratively generate each column. In each iteration, we first append both the column name and data type to *stmt* (Lines 5–6), and then append column constraints, e.g., `UNIQUE` and `PRIMARY KEY`, according to the column key type, e.g., `UNI` and `PRI`, respectively (Lines 7–10). Furthermore, if there are `FOREIGN KEY` constraints, we append the foreign key definition including the source column and the associated reference table and column to *stmt* (Lines 14–15). For other column and table attributes, e.g., `DEFAULT` and `TEMPORARY`, we apply the same method to append them. Note that the process of constructing *createView* statements is similar to Algorithm 2.

After generating a sequence of `CREATE` statements, we order them based on their dependencies, i.e., `FOREIGN KEY` constraints. Specifically, we first create tables that do not have `FOREIGN KEY` constraints. Next, given a `FOREIGN KEY` constraint that contains a source table *srcTable* and a reference table *refTable*, *refTable* should be created before *srcTable*. Finally, we create other kinds of database objects if necessary, e.g., views.

Circular dependencies occur when two or more tables are interdependent due to their `FOREIGN KEY` constraints. If the database schema has circular dependencies, we cannot synthesize the `CREATE TABLE` statements by using Algorithm 2, since we cannot

Algorithm 3: *schemaBug* Detection

Input: The generated DDL sequence seq_{gen}, the synthesized DDL sequence seq_{syn}

1 **Function** validateSequence(seq_{gen}, seq_{syn}) **do**
2 $db_{gen} \leftarrow$ execute(seq_{gen})
3 $db_{syn}, error \leftarrow$ execute(seq_{syn})
4 **if** $error! = NULL$ **then**
5 **report** *schemaBug*
6 **for** $i \leftarrow 1$ **to** *maxStatements* **do**
7 $stmt \leftarrow$ generateSQLstatement(db_{gen})
8 $error_1, result_1 \leftarrow$ execute($db_{gen}, stmt$)
9 $error_2, result_2 \leftarrow$ execute($db_{gen}, stmt$)
10 **if** $result_1 \neq result_2$ || $error_1 \neq error_2$ **then**
11 **report** *schemaBug*

reference a non-existing table during table creation. To address this issue, DDLCheck first constructs CREATE TABLE statements to create the same database objects except for their FOREIGN KEY constraints by using Algorithm 2. Then DDLCheck constructs ALTER TABLE ADD FOREIGN KEY statements, creating the same FOREIGN KEY constraints.

3.4 *schemaBug* Detection

After generating a complex DDL sequence seq_{gen} and its corresponding synthesized DDL sequence seq_{syn}, we detect *schemaBug*s through the process shown in Algorithm 3. First, we execute seq_{gen} and seq_{syn} to create the equivalent databases db_{gen} and db_{syn} (Lines 2–3). The synthesized DDL sequences are expected to execute without errors. If an error is returned, DDLCheck reports a *schemaBug* (Line 5). In such cases, the target DBMS stores incorrect schema-related information for the generated database (e.g., wrong information in the system table innodb_table_stats in MariaDB), which is then used to generate the synthesized DDL sequence. Otherwise, we will check whether the equivalent databases db_{gen} and db_{syn} return the same execution results and database states for the same SQL statements.

Based on the type of $stmt$, we apply different ways to obtain and validate its execution results (Lines 7–11). Specifically, if $stmt$ is a DML statement, we construct SELECT statements to retrieve the table data $result_1$ and $result_2$. If $stmt$ is a SELECT statement, we obtain the returned query results $result_1$ and $result_2$. We also collect error messages (i.e., $error_1$ and $error_2$) returned by the target DBMS during executing $stmt$ (Lines 8–9). If the target DBMS returns different results or different error messages, DDLCheck reports a *schemaBug* (Lines 10–11).

DDLCheck continuously generates *maxStatements* statements to test databases db_{gen} and db_{syn}, which is configured to 5,000 in our experiments. Once a *schemaBug* is reported, i.e., the equivalent databases db_{gen} and db_{syn} return inconsistent execution results for a specific statement $stmt$, DDLCheck stops testing.

Based on the execution results of $stmt$, we can identify the following three types of *schemaBug*s:

Algorithm 4: Translating DDL Sequences to Database Schema Transition Graphs

Input : A DDL sequence $seq = \{s_1, s_2, \ldots, s_n\}$
Output: A database schema transition graph G

1 $G \leftarrow \emptyset$ // Initialize an empty graph
2 $schema \leftarrow \emptyset$ // Start with an empty database schema
3 **foreach** $s_i \in seq$ **do**
4 $schema \leftarrow$ updateSchema($schema, s_i$)
5 $ddlType \leftarrow$ getDDLType(s_i)
6 **if** $ddlType$ *is* CREATE TABLE **then**
7 $rNode \leftarrow G.addNode(s_i)$ // A root node
8 $nTable \leftarrow$ getDefinedTable($schema, s_i$)
9 $nNode \leftarrow G$.addNode($nTable$)
10 G.addRelationship($rNode, nNode, s_i$)
11 **else if** $ddlType$ *is* DROP TABLE **then**
12 $mNode \leftarrow$ getNode(G, s_i)
13 $nNode \leftarrow G$.addNode($NULL$)
14 G.addRelationship($mNode, nNode, s_i$)
15 **else**
16 $curTable \leftarrow$ getModifiedTable($schema, s_i$)
17 $nNode \leftarrow G$.addNode($curTable$)
18 $mNode \leftarrow$ getNode(G, s_i)
19 G.addRelationship($mNode, nNode, s_i$)
20 **return** G

- **Incorrect database schema**. This type of *schemaBug*s occurs when the execution of the DDL sequence creates incorrect database schemas or throws unexpected errors.
- **Incorrect database state**. This type of *schemaBug*s occurs when the equivalent databases db_{gen} and db_{syn} store different table content after executing $stmt$.
- **Incorrect query result**. This type of *schemaBug*s occurs when the equivalent databases db_{gen} and db_{syn} return different query results for $stmt$.

3.5 Sequence-Oriented Testing Optimization

Different DDL sequences can create the same or similar schema-related information. This situation can reduce the testing efficiency of DDLCheck. Testing on these DDL sequences usually triggers duplicate *schemaBug*s, and can hardly reveal new *schemaBug*s. To avoid testing these duplicate sequences, we design a sequence-oriented testing optimization strategy, in which a database schema transition graph (DSTG) is maintained to record the table structure change history conducted by the DDL sequence. In the following, we first illustrate the DSTG translation and then present the process of selecting interesting DDL sequences.

3.5.1 Database Schema Transition Graph. A database schema transition graph (DSTG) is used to represent the structure changes of each table in the generated database schema conducted by a DDL sequence. We model a DSTG as a graph $G = (N, R, \rho)$, where:

- N is a finite set of nodes, and each node n stores the structure $struct$ of a table (i.e., $n.struct$).

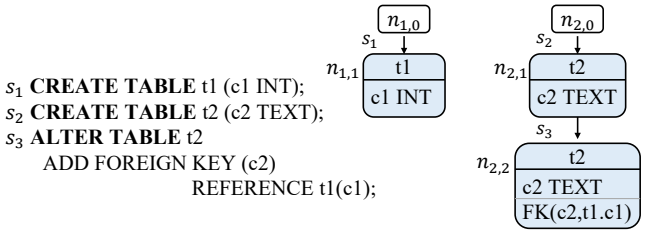

Figure 3: An example of database schema transition graph.

- R is a finite set of relationships, and each relationship r contains a DDL statement $ddlStmt$ (i.e., $r.ddlStmt$).
- $\rho : R \rightarrow N \times N$ is a function that maps a relationship r to its incoming node $inNode$ and outgoing node $outNode$, in which applying $r.ddlStmt$ on $inNode.struct$ can generate $outNode.struct$.

We use a DSTG to record the evolution of table structure history conducted by the DDL sequences. Algorithm 4 presents the process of translating a given DDL sequence into the DSTG. Specifically, given an input DDL sequence $seq = \{s_1, s_2, \ldots, s_n\}$, we first initialize an empty graph G (Line 1). Starting from an empty database schema $schema$ (Line 2), we traverse the sequence seq and update the DSTG based on the type $ddlType$ of each statement s_i, e.g., `CREATE TABLE`, `CREATE INDEX`, and `DROP TABLE` (Lines 4–19). If s_i is a `CREATE TABLE` statement, we add a root node and append it with a node to store the newly created table (Lines 7–10). If s_i is a `DROP TABLE` statement, we retrieve the corresponding modified node $mNode$ and link $mNode$ to an empty node $nNode$ with the value `NULL` (Lines 12–14). If s_i is other type of DDL statement, e.g., `CREATE INDEX` and `ALTER TABLE`, we first obtain the table structure $curTable$ after executing s_i and store $curTable$ into a new node $nNode$ (Lines 16–17). We then link $nNode$ to its corresponding modified node $mNode$ (Lines 18–19).

Note that if the table is created with a `FOREIGN KEY` constraint or is modified by adding a `FOREIGN KEY` constraint, we also store the reference table structure in the corresponding source table structure. For example, given a table `t1` with a `FOREIGN KEY` constraint *FOREIGN KEY (c1) REFERENCES t2 (c2)*, when storing the table structure `t1`, we also store the table structure of table `t2`.

Figure 3 shows an example DSTG that is translated by the DDL sequence $seq = \{s_1, s_2, s_3\}$. We first execute the `CREATE TABLE` statement s_1 and create a node $n_{1,1}$ to store the defined table structure of $t1$. Then we execute the `CREATE TABLE` statement s_2 and create a node $n_{2,1}$ to store the defined table structure of $t2$. Next, we execute the `ALTER TABLE` statement s_3 to change the table structure defined in $n_{2,1}$, and a node $n_{2,2}$ is created to store the updated table structure of $t2$. Finally, we obtain the DSTG G with five nodes $N = \{n_{1,0}, n_{1,1}, n_{2,0}, n_{2,1}, n_{2,2}\}$ and three relationships.

3.5.2 Choosing Interesting DDL Sequences. We record all the tested DSTGs in a set $UniqueSequence$. After generating a DDL sequence seq_{gen}, we apply Algorithm 4 to translate seq_{gen} into a DSTG G. If the set $UniqueSequence$ contains a similar DDL sequence whose DSTG structure is equivalent to G, we do not test the sequence seq_{gen} but generate a new DDL sequence.

$$\texttt{table} := \langle tName, column+, index*, foreignKey* \rangle$$
$$\texttt{column} := \langle cName, type, constraint* \rangle$$
$$\texttt{constraint} := \text{NOT NULL} \mid \text{PRIMARY KEY} \mid \text{UNIQUE} \mid \ldots$$
$$\texttt{index} := [\text{UNIQUE}](cName+)$$
$$\texttt{foreignKey} := \text{FOREIGN KEY}(srcCol, targetCol)$$

Figure 4: A formal description of the table structure. $term+$ (e.g., $column+$) denotes one or more terms, and $term*$ (e.g., $index*$) denotes zero or more terms.

We define that two DSTGs $G_1 = (N_1, R_1, \rho_1)$ and $G_2 = (N_2, R_2, \rho_2)$ are equivalent ($equal(G_1, G_2)$ for short), if they satisfy the following three conditions.

- For each node $n_1 \in N_1$, we can find a corresponding node $n_2 \in N_2$ that stores the equivalent table structure (i.e., $equal(n_1.struct, n_2.struct)$).
- For each relationship $r_1 \in R_1$, we can find a corresponding relationship $r_2 \in R_2$ that has the same DDL type (i.e., $r_1.ddlStmt.ddlType = r_2.ddlStmt.ddlType$). Note that $r_1.ddlStmt$ and $r_2.ddlStmt$ can be different, since we only check the equivalence of two table structures. Thus, we only care about the types of DDL statements.
- For each relationship $r_1 \in R_1$, we can find a corresponding relationship $r_2 \in R_2$ that contains the equivalent incoming and outgoing nodes as r_1, i.e., $equal(\rho_1(r_1).inNode.struct, \rho_2(r_2).inNode.struct)$, and $equal(\rho_1(r_1).outNode.struct, \rho_2(r_2).outNode.struct)$.

Figure 4 presents a formal description of the table structure. We define that two tables $t1$ and $t2$ have the *equivalent table structure* if and only if there exists a bijective mapping f from the columns of $t1$ to those of $t2$, satisfying the following conditions:

- **Columns**: $t1$ and $t2$ have the same number of columns. For each column c in $t1$, the data type of c and the set of constraints on c match those of $f(c)$ in $t2$.
- **Indexes**: $t1$ and $t2$ have the same number of indexes. Each index in $t1$ corresponds to an index in $t2$ that has identical columns as defined by f and the same `UNIQUE` constraint.
- **Foreign Keys**: $t1$ and $t2$ have the same set of `FOREIGN KEY` constraints. Each `FOREIGN KEY` constraint in $t1$ maps to a `FOREIGN KEY` constraint in $t2$ such that the source and target columns correspond according to f.

Note that if two tables $t1$ and $t2$ contain multiple columns that have the same data type and the same set of constraints, we maintain multiple bijective mappings among these columns. If there exists a mapping that can be used to map indexes and `FOREIGN KEY` constraints, we consider that the two tables $t1$ and $t2$ have the same table structure.

Figure 5 shows that the two tables $t1$ and $t2$ have the same table structure, in which we can find a column mapping where $f(t1.c1) = t2.c3$, $f(t1.c2) = t2.c1$, and $f(t1.c3) = t2.c2$. Since all columns and indexes can be mapped in this way according to f, $t1$ and $t2$ have the same table structure.

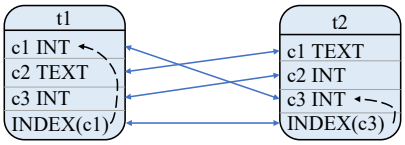

Figure 5: Tables $t1$ and $t2$ have the same table structure.

3.6 SQL Statement Generation

SQL statement generation has been widely explored in previous researches [9, 11, 38, 44, 63] and is not a contribution of our work. We briefly present our SQL statement generation as follows.

We use a grammar-based approach to generate SQL statements, building a grammar model for each type of SQL statement. We traverse the grammar model to generate different SQL statements. Specifically, we define a set of production rules for each type of SQL statement to capture its syntactic structure and constraints. These production rules are used to generate a parse tree, representing a specific SQL statement. By traversing the parse tree, we can generate the corresponding SQL statement in string format. Since SQL statement generation is a stateful process, the database schema is dynamically maintained to provide valid semantic information, such as table and column names.

DDLCheck supports various types of DML statements, e.g., `INSERT`, `UPDATE`, and `DELETE`. For `UPDATE` and `DELETE` statements, complex predicates (i.e., `WHERE` clauses) can be generated. For `SELECT` statements, most syntax features supported by the target DBMS (e.g., joins, sub-queries, window functions, and complex predicates) can be generated. However, DDLCheck does not support the following syntax features: (1) non-deterministic and time-related functions, e.g., the `Rand` function returns a different value each time it is called; (2) ambiguous queries that yield different values across executions.

4 EVALUATION

We implement DDLCheck on our target DBMSs with 4,828 lines of Java code. DDLCheck primarily consists of five functionalities, including generating DDL sequences, DDL-sequence-oriented testing optimization, synthesizing DDL sequences, generating DML and DQL statements and detecting *schemaBug*s.

We evaluate the effectiveness of DDLCheck by investigating the following three research questions:

- **RQ1**. What *schemaBug*s can DDLCheck detect in real-world relational DBMSs?
- **RQ2**. How effective is the DDL-sequence-oriented testing optimization strategy in DDLCheck?
- **RQ3**. How many bugs detected by DDLCheck can be found by existing approaches?

4.1 Experimental Setup

Target DBMSs. We select six widely-used relational DBMSs to evaluate the effectiveness of DDLCheck. These DBMSs are described in Table 1. Specifically, during the development of our prototype, we test the latest release versions available at that time, namely MySQL 8.0.36, PostgreSQL 16.2, SQLite 3.43.0, MariaDB 11.3.2, CockroachDB 23.2 and TiDB 8.0.0. To find more unique and new *schemaBug*s, we continuously monitor for updates and test newer versions of these DBMSs.

Experimental infrastructure. We conduct our experiments on a machine with 8 CPU cores and 32GB RAM. We follow the official documentation for each DBMS to deploy and configure them correctly. Specifically, we utilize Docker containers to deploy MySQL, MariaDB, and PostgreSQL, while CockroachDB and TiDB are deployed using their official binary files. Since SQLite is an embedded DBMS, we do not need to independently deploy it.

Testing methodology. We run DDLCheck on each target DBMS for 24 hours, and then stop it to analyze the generated bug reports. For the generated DDL sequence seq_{gen} and its corresponding synthesized DDL sequence seq_{syn}, DDLCheck continuously generates a specified number $maxStatements$ of statements to test their created databases db_{gen} and db_{syn}, which is set to 5,000 in our experiment[1].

If a DBMS returns different execution results for a statement $stmt$ executed on seq_{gen} and seq_{syn}, DDLCheck reports a *schemaBug*. We refer to $stmt$ as the bug-revealing statement. The bug report consists of the sequences seq_{gen} and seq_{syn}, along with the testing SQL statements T, which we refer to as the collection of SQL statements generated prior to $stmt$. Note that we also record the execution results of $stmt$, including query results for `SELECT` statements, database states for DML statements, and any error messages.

For each generated bug report, we first automatically simplify it and then check whether this bug has been reported previously. For a crash bug, we check for the existence of another crash bug with the same stack trace. For a *schemaBug*, we check for the existence of another bug with similar syntactic features. If a similar bug is found, we do not submit this bug report to avoid reporting duplicates.

We implement two bug report reduction techniques to automatically simplify the bug reports generated in DDLCheck. Our bug report reduction techniques focus on simplifying the generated sequence seq_{gen}, the testing statements T, and the bug-revealing statement $stmt$. The simplified seq_{syn} can be synthesized from the simplified seq_{gen}.

For the generated sequence seq_{gen}, we first apply a statement-based reduction technique by removing unnecessary DDL statements that do not contribute to revealing the bug. Specifically, we randomly remove some DDL statements and check whether the bug still occurs. If the bug occurs after removing the statements, we continue to remove other statements until no statement can be removed. Otherwise, we keep them and try to remove other statements. After that, we apply a syntax-based reduction technique to simplify the remaining statements. Specifically, we parse each statement to get the abstract syntax tree (AST) and traverse the AST to randomly remove optional attributes and operations. For each step, we translate the AST into a statement and check whether the bug still occurs. If so, we continuously remove unnecessary attributes and operations until no attribute or operation can be removed. Otherwise, we keep them and remove other attributes and operations.

[1]We first investigate the number of statements generated by existing DBMS testing approaches, e.g., 5,000 in Radar [52], 25,000 in Pinolo [31], and 100,000 in SQLancer [14, 15, 47–49]. Then we run DDLCheck on our target DBMSs with setting $maxStatements$ as 5,000, 25,000, and 100,000 for 24 hours, respectively. Our experimental results show that the value 5,000 of $maxStatements$ is more suitable for DDLCheck to detect more unique bugs quickly. Therefore, we set $maxStatements$ to an empirical value 5,000 in our experiment.

Table 2: Overall detection results for DDLCheck

DBMS	Bug Status					Bug Categories	
	Submitted	Confirmed	Fixed	Duplicate	False Positive	*schemaBug*	Crash
MySQL	14	14	2	0	0	13	1
SQLite	0	0	0	0	0	0	0
PostgreSQL	0	0	0	0	0	0	0
MariaDB	11	7	3	3	0	5	2
CockroachDB	2	1	0	1	0	1	0
TiDB	7	7	4	0	0	2	5
Total	**34**	**29**	**9**	**4**	**0**	**21**	**8**

For the testing statements T, we apply the same statement-based and syntax-based reduction techniques as those in simplifying seq_{gen}. For the bug-revealing statement $stmt$, we apply the same syntax-based reduction technique as in simplifying seq_{gen}.

We also design an object-based reduction technique to simplify the generated sequence seq_{gen} and the testing statement T, since complex dependencies among SQL statements hurt the effectiveness of the statement-based reduction technique. The object-based reduction technique simplifies the database created by seq_{gen} and T. Specifically, we first construct `DROP` and `ALTER` statements to remove unnecessary database objects (e.g., tables and columns). Then we export data as `INSERT` statements and remove unnecessary `INSERT` statements. After simplifying the database, we synthesize the corresponding DDL sequence as described in Section 3.3.

Given a bug report, we first use the statement-based and syntax-based technique to simplify it. If the bug-revealing statement is a `SELECT` statement, we analyze its execution plan to add suitable query hints [15, 30, 56]. In such way, we can easily reproduce the bug and use the object-based reduction technique to simplify the bug report. Otherwise, we only use the former reduction technique and provide the execution plans of `SELECT` statements to DBMS developers for further analysis.

4.2 Overall Detection Results

We investigate RQ1 to evaluate the effectiveness of `DDLCheck` in detecting *schemaBug*s in real-world relational DBMSs. We continuously test our target DBMSs and analyze the generated bug reports by following the method discussed in Section 4.1. Table 2 shows the overall bug detection results for `DDLCheck`. We have submitted 34 bugs to DBMS developers, including 14 bugs in MySQL, 11 bugs in MariaDB, 2 bugs in CockroachDB, and 7 bugs in TiDB. In our experiments, we have not yet found new bugs in PostgreSQL and SQLite.

Bug status. Among the 34 bugs, 29 bugs have been confirmed as new bugs, and 9 bugs have been fixed by the corresponding DBMS developers. For the remaining 5 bugs, one bug in MariaDB is open, and 4 bugs are duplicate to existing bug reports including 3 bugs in MariaDB and one bug in CockroachDB.

Bug severity. Among the 29 confirmed bugs, 23 bugs have the bug severity of *Critical* or *Major*, including 9 bugs in MySQL, 7 bugs in MariaDB, and 7 bugs in TiDB. For the remaining 5 bugs, 4 bugs in MySQL have a bug severity of *Noncritical*, and 1 bug in CockroachDB is not labelled with bug severity. The experimental result shows that most of the bugs found by `DDLCheck` are considered critical by the corresponding DBMS developers.

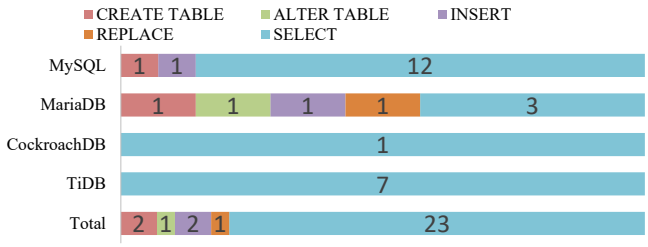

Figure 6: The distribution of bug-revealing statements.

Bug categories. Among the 29 confirmed bugs, 21 bugs are *schemaBug*s, including 13 bugs in MySQL, 5 bugs in MariaDB, 1 bug in CockroachDB, and 2 bugs in TiDB. The remaining 8 bugs are crash bugs, including 1 bug in MySQL, 2 bugs in MariaDB, and 5 bugs in TiDB. Since crash bugs cause explicit errors, `DDLCheck` can also detect crash bugs.

Bug consequences. Among the 21 confirmed *schemaBug*s, 3 bugs cause *incorrect database schema*, 4 bugs cause *incorrect database state*, and 14 bugs cause *incorrect query result*. The remaining 8 crash bugs shut down the DBMS server.

4.3 Bug Analysis

In this section, we further analyze the 29 confirmed bugs, including their bug manifestations and root causes.

4.3.1 Bug Manifestation. We first analyze the bug-revealing statements and DDL options of our 29 confirmed bugs, and then analyze the complexity of DDL sequences that trigger *schemaBug*s.

Bug-revealing statements. Figure 6 shows the distribution of bug-revealing statements in our 29 confirmed bugs. DDL statements reveal 3 bugs, in which `CREATE TABLE` statements reveal 2 bugs and `ALTER TABLE` statements reveal 1 bug. DML statements reveal 3 bugs, in which `INSERT` statements reveal 2 bugs and `REPLACE` statements reveal 1 bug. `SELECT` statements reveal the remaining 23 bugs. The experimental result shows that database schema evolution significantly affects the execution of `SELECT` statements.

DDL options. 5 out of 29 confirmed bugs require specific options that are used to fine-tune DBMS performance, including KEY_BLOCK_SIZE, STATS_PERSISTENT, ROW_FORMAT, and ALGORITHM. Note that two bugs require the ALGORITHM option. The remaining 24 confirmed bugs do not require such options.

DDL sequences. In our test oracle, we construct a complex DDL sequence that consist of various types of DDL statements and a simple synthesized DDL sequence to evaluate the same SQL statements

in equivalent databases. Among 21 *schemaBugs*, 20 *schemaBugs* are triggered by complex DDL sequences, while only one *schemaBug* is triggered by the synthesized DDL sequence (i.e., MySQL#114109 in Listing 2). For the remaining 8 crash bugs, both complex DDL sequences and simple synthesized DDL sequences can trigger all of them. The experimental results show that both complex DDL sequences and simple synthesized sequences can trigger *schemaBugs*, and complex DDL sequences are more likely to trigger *schemaBugs*.

```
1. -- Sequence seq_gen
2. CREATE TABLE t0 (c0 SMALLINT STORAGE DISK UNIQUE) ENGINE=
     MyISAM;
3. ALTER TABLE t0 KEY_BLOCK_SIZE=63705;

4. -- Sequence seq_syn
5. CREATE TABLE t0 (c0 SMALLINT STORAGE DISK UNIQUE) KEY_BLOCK_SIZE=63705,
     ENGINE=MyISAM; - Crash 🐞
```

Listing 2: MySQL#114109. Executing the synthesized `CREATE` statement causes the MySQL server to crash down.

> *Complex DDL sequences are more likely to trigger schemaBugs. This indicates that the database community should pay more attention to the quality of database schema evolution.*

4.3.2 Root Cause. We further analyze the root causes of our 29 confirmed bugs. For 14 of these bugs, we can clearly identify their root causes, which can be categorized into three types: (1) incorrect maintenance of database schemas, (2) incorrect maintenance of schema-related information, and (3) incorrect query optimization. Specifically, we identified that 1 bug was caused by incorrect maintenance of database schemas, 4 bugs were caused by incorrect maintenance of schema-related information, and 9 bugs were caused by incorrect query optimization. For the remaining 15 bugs, we do not have enough information to identify their root causes. In the following, we present the detailed analysis of five representative bugs with their identified root causes.

```
1. -- Sequence seq_gen
2. CREATE TABLE t0 (c2 INT, PRIMARY KEY (c2));
3. CREATE TABLE t1 (c1 INT UNIQUE, FOREIGN KEY (c1) REFERENCES
     t0(c2));
4. ALTER TABLE t0 RENAME t2, ALGORITHM COPY; 🐞
5. -- Sequence seq_syn
6. CREATE TABLE t2 (c2 INT, PRIMARY KEY (c2));
7. CREATE TABLE t1 (c1 INT UNIQUE, FOREIGN KEY (c1) REFERENCES
     t0(c2)); -- Throw an error: Foreign key constraint is
     incorrectly formed
```

Listing 3: MDEV#34105. MariaDB stores incorrect database schemas in system tables `INFORMATION_SCHEMA`.

Incorrect maintenance of database schemas. Listing 3 shows a *schemaBug* MDEV#34105, which describes an inconsistency in MariaDB's system tables, specifically in the `INFORMATION_SCHEMA`. The root cause of this *schemaBug* is that MariaDB does not propagate changes in table names to the `INFORMATION_SCHEMA` system tables when a table involved in a `FOREIGN KEY` constraint is renamed (Lines 2–4). As a result, the stored database schema remains inconsistent, in which the foreign key constraint in `t1` continues to reference `t0`, which no longer exists, instead of being updated to reference `t2`. When attempting to recreate the schema (as shown in seq_{syn}), MariaDB returns a "Foreign key constraint is incorrectly formed" error because it tries to enforce a foreign key constraint on a non-existent table (Lines 6–7).

```
1. -- Sequence seq_gen
2. CREATE TABLE t0 (c2 INT PRIMARY KEY);
3. CREATE TABLE t1 (c1 INT UNIQUE, FOREIGN KEY (c1) REFERENCES
     t0(c2));
4. ALTER TABLE t0 RENAME t2, ALGORITHM COPY; 🐞
5. -- Sequence seq_syn
6. CREATE TABLE t2 (c2 INT PRIMARY KEY);
7. CREATE TABLE t1 (c1 INT UNIQUE, FOREIGN KEY (c1) REFERENCES
     t2(c2));
8. -- Test statements T
9. INSERT INTO t2 VALUE (1);
10. INSERT INTO t1 VALUE (1);
11. -- db_gen's state: t1:{}, t2:{1} ✘
12. -- db_syn's state: t1:{1}, t2:{1} ✔
```

Listing 4: MySQL#114904. MySQL does not promptly refresh schema-related information after renaming a table.

Incorrect maintenance of schema-related information. Listing 4 shows a bug MySQL#114904, in which MySQL does not promptly refresh schema-related information after renaming a table that is referenced by a `FOREIGN KEY` constraint (Lines 2–4). The system tables and schema-related information still reference `t0`, which no longer exists, rather than the new table name `t2`. This wrong behavior causes data insertion to fail due to a `FOREIGN KEY` constraint violation (Line 11), even though the inserted data logically satisfy the constraint. In the synthesized sequence seq_{syn}, since `t1` directly references `t2(c2)` (Lines 6–7), the `FOREIGN KEY` constraint is correct, and subsequent statements proceed without any issues (Line 12). That is the stored database schema is correct.

```
1. -- Sequence seq_gen
2. CREATE TABLE t1 (c1 INT);
3. CREATE TABLE t2 (c1 INT);
4. ALTER TABLE t1 STATS_PERSISTENT 0; 🐞
5. DROP TABLE IF EXISTS t1;
6. ALTER TABLE t2 RENAME t1; -- Throw an error: duplicate key
     in table 'innodb_table_stats'
7. -- Sequence seq_syn
8. CREATE TABLE t1 (c1 INT) STATS_PERSISTENT=0;
9. CREATE TABLE t2 (c1 INT);
10. DROP TABLE IF EXISTS t1;
11. ALTER TABLE t2 RENAME t1; -- No Errors
```

Listing 5: MDEV#34207. MariaDB stores incorrect table statistics in system table `innodb_table_stats`.

Listing 5 shows another bug MDEV#34207 that is caused by incorrect maintenance of table statistics. In this bug, MariaDB fails to properly update entries in the `innodb_table_stats` system table after executing the *ALTER TABLE t1 STATS_PERSISTENT 0* statement (Lines 2–4). As a result, the lingering entry causes a conflict if another table is later renamed to the dropped table's name, because the statistics system detects a duplicate entry (Lines 5–6). When the option *STATS_PERSISTENT* is set to 0 during table creation, MariaDB properly clears these entries, allowing renaming statements can proceed without encountering duplicate key conflicts (Lines 8–11).

The bug MDEV#35115 shown in Listing 6 is caused by inconsistent index orders. In this bug, MariaDB stores inconsistent orders of secondary indexes in the InnoDB structure and the table structure of `t1`. This inconsistent index order affects the execution of `REPLACE` statements, which rely on `UNIQUE` constraints to locate rows for replacement. As a result, MariaDB fails to accurately enforce the `UNIQUE` constraints, resulting in unexpected errors and incorrect database states (Line 10). When the index is defined during

table creation (Line 5), MariaDB executes the REPLACE statement correctly (Line 11).

```
1. -- Sequence seq_gen
2. CREATE TABLE t1 (c1 NUMERIC UNSIGNED NOT NULL, c2 INT3
     UNIQUE, c3 BIT(2) PRIMARY KEY) Engine=InnoDB;
3. CREATE UNIQUE INDEX i1 ON t1(c1); 🐞
4. -- Sequence seq_syn
5. CREATE TABLE t1 (c1 NUMERIC UNSIGNED NOT NULL UNIQUE, c2
     INT3 UNIQUE, c3 BIT(2) PRIMARY KEY);

6. -- Test statements T
7. INSERT INTO t1 (c1,c2,c3) VALUES (0,0,b'01');
8. INSERT INTO t1 (c1,c2,c3) VALUES (1,1,b'10');
9. REPLACE INTO t1 (c1,c2,c3) VALUES (0,1,b'11');
10. -- db_gen's state: t1:{(0,0,b'01'), (1,1,b'10')} ✘
11. -- db_gen's state: t1:{(0,1,b'11')} ✔
```

Listing 6: MDEV#35115. Inconsistent replace behavior when multiple unique indexes exist.

Besides the above three *schemaBug*s that are caused by incorrect maintenance of schema-related information, Listing 1 is also attributed to this root cause. Since we have carefully introduced it earlier, we do not discuss it here.

```
1. -- Sequence seq_gen
2. CREATE TABLE t1 (c1 BIT);
3. CREATE TABLE t2 (c1 TEXT);
4. CREATE INDEX i1 ON t1(c1 ASC);

5. -- Sequence seq_syn
6. CREATE TABLE t1 (c1 BIT, UNIQUE i1 (c1 ASC));
7. CREATE TABLE t2 (c1 TEXT);

8. -- Test statements T
9. INSERT INTO t1 VALUES (0);
10. INSERT INTO t2 VALUES ('0');
11. SELECT t1.c1, t2.c1 FROM t1 NATURAL JOIN t2;
12. -- {0} in db_gen ✔
13. -- {} in db_syn ✘
```

Listing 7: MySQL#114539. The SELECT statement returns different query results.

Incorrect query optimization. Listing 7 shows a bug MySQL#114539, in which the SELECT statement returns different query results (Lines 11–13). The database db_{gen} is created by the sequence seq_{gen}, in which we first create two tables t1 and t2 and then build an index $i1$ on table t1 (Lines 2–4). In database db_{syn}, the index is created during table creation (Lines 6–7). For databases db_{gen} and db_{syn}, MySQL returns inconsistent execution results (Lines 12–13), in which db_{gen} returns {0} but db_{syn} returns {}. We obtain the execution plans for the SELECT statement, in which db_{gen} joins two tables t1 and t2 without the index $i1$, but db_{syn} performs the same join operation with the index $i1$. Note that the optimizer hint $/* + NO\_INDEX(t1) */$ can be used to reproduce the buggy execution of the SELECT statement.

When storing or updating database schemas, DBMS developers should take care about the consistency of multiple schema-related information.

4.4 Effectiveness of DDL-Sequence-Oriented Testing Optimization Strategy

To evaluate the effectiveness of the DDL-sequence-oriented testing optimization strategy, we answer RQ2 by designing two variants of the optimization strategy and comparing them with DDLCheck. Specifically, DDLCheck$_{rand}$ does not use the DDL-sequence-oriented testing optimization strategy of DDLCheck and tests all the generated DDL sequences. DDLCheck$_{strict}$ adopts a more strict strategy than DDLCheck by enforcing the same column order when comparing two table structures (Section 3.5.2). We run them on our target DBMSs for 24 hours. During these experiments, we count the number of generated DDL sequences, unique DDL sequences, total bugs, and unique bugs.

Table 3 demonstrates that DDLCheck can generate and test more unique DDL sequences than DDLCheck$_{rand}$ and DDLCheck$_{strict}$ for the same amount of time. Specifically, for the same amount of time, DDLCheck generates 8,338 more DDL sequences and tests 5,474 more unique DDL sequences than DDLCheck$_{rand}$. Meanwhile, DDLCheck generates 4,178 more DDL sequences and tests 2,501 more unique DDL sequences than DDLCheck$_{strict}$.

Furthermore, DDLCheck can also detect more unique *schemaBug*s than DDLCheck$_{rand}$ and DDLCheck$_{strict}$ for the same amount of time. Specifically, for the same amount time, DDLCheck detects 3 more unique bugs than DDLCheck$_{rand}$. Meanwhile, DDLCheck detects 4 more unique bugs than DDLCheck$_{strict}$. These experimental results also indicate that our testing optimization strategy does not exclude interesting sequences that lead to bugs.

To further investigate whether the excluded sequences (by ignoring column orders) could lead to missing bugs, we investigated our confirmed 29 bugs by changing their column orders and checking whether all possible column orders can trigger the same bugs. We found that all possible column orders can trigger these 29 bugs. This indicates that ignoring column orders will not miss any of the 29 confirmed bugs.

Our proposed DDL-sequence-oriented testing optimization strategy can help DDLCheck *test more unique DDL sequences and detect unique schemaBugs more quickly.*

4.5 Comparing with Existing Approaches

We answer RQ3 by comparing DDLCheck with three state-of-the-art approaches, namely NoREC [47], Radar [52], and DQP [15], since they are designed to detect optimization bugs in SELECT statements and have thoroughly tested our target DBMSs. In this experiment, we investigate whether the 21 confirmed *schemaBug*s can be detected by these approaches. Since crash bugs are explicit errors, we assume that these approaches can also detect them. Our conceptual comparison result demonstrates the effectiveness of DDLCheck in detecting *schemaBug*s (as shown in Table 4).

NoREC constructs two equivalent SELECT statements by transforming the WHERE clause for a given SELECT statement. Given a bug report containing the database *db* and the bug-revealing statement *stmt*, if *stmt* is a DDL or DML statement or a SELECT statement without WHERE clauses, NoREC cannot perform this transformation and thus cannot detect the bug. If *stmt* is a SELECT statement with a WHERE clause, we perform the transformation and compare their query results. Although seven bugs contain a SELECT statement with a WHERE clause, the WHERE clauses in these statements contain sub-queries that NoREC does not support. As a result, NoREC cannot detect all 21 confirmed *schemaBug*s.

Radar removes data constraints and indexes from a database and observes whether this transformation changes the query result

Table 3: Comparison of bug detection between DDLCheck (Ori), $\texttt{DDLCheck}_{rand}$ (Rand), and $\texttt{DDLCheck}_{strict}$ (Strict)

	Generated Sequences			Unique Sequences			Total Bugs			Unique Bugs		
DBMS	**Ori**	**Rand**	**Strict**	**Ori**	**Rand**	**Strict**	**Ori**	**Rand**	**Strict**	**Ori**	**Rand**	**Strict**
MySQL	3599	2423	3057	3037	2041	2585	5	4	6	4	3	3
SQLite	6720	4340	5249	4819	3164	3792	0	0	0	0	0	0
PostgreSQL	6385	3640	5031	3657	2418	3236	0	0	0	0	0	0
MariaDB	5010	3659	4697	4222	3114	3986	2	1	3	2	1	1
CockroachDB	3211	2894	2890	2541	2261	2293	1	1	0	1	1	0
TiDB	2664	2395	2487	2198	1996	2075	1	0	0	1	0	0
Total	**27589**	**19251**	**23411**	**20468**	**14994**	**17967**	**9**	**6**	**9**	**8**	**5**	**4**

Table 4: Conceptual comparison with existing approaches

DBMS	DDLCheck	NoREC	Radar	DQP
MySQL	13	0	10	8
MariaDB	5	0	1	0
CockroachDB	1	0	1	0
TiDB	2	0	1	1
Total	**21**	**0**	**13**	**9**

for a given SELECT statement. Given a bug report containing the database *db* and the bug-revealing statement *stmt*, if *stmt* is a DDL or DML statement, Radar cannot detect the bug. If *stmt* is a SELECT statement, we first check whether *db* contains data constraints and indexes. If *db* contains data constraints and indexes, we remove them. Otherwise, we add data constraints and indexes and observe whether *stmt* returns a different query result. As a result, Radar detects only 13 of the 21 confirmed *schemaBug*s.

DQP enumerates possible execution plans for a given SELECT statement by applying query hints or setting system variables that affect the query optimizer. Given a bug report containing the database *db* and the bug-revealing statement *stmt*, if *stmt* is a DDL or DML statement, DQP cannot detect the bug. If *stmt* is a SELECT statement, we first check whether the DBMS executes *stmt* in different execution plans and then attempt to add query hints or set system variables. If we can add query hints to *stmt* or set system variables to reproduce the bug, we assume that DQP can theoretically detect it. As a result, DQP can detect only 9 out of the 21 confirmed *schemaBug*s.

5 RELATED WORK

SQL statement generation. Generating SQL statements is a fundamental part of testing DBMSs, in which test cases are a set of SQL statements. SQL statement generation has been widely explored by existing works [9, 11, 13, 16, 17, 25, 26, 32, 44, 45, 57, 63]. SQLsmith [11] and SQLancer [9] internally maintain a grammar model to randomly generate SQL statements. Squirrel [63] and SQLRight [44] apply a mutation-based method to generate SQL statements using a designed intermediate representation. Unlike the above approaches, which require a precise understanding of SQL grammar, Griffin [26] maintains semantic relations and shuffles existing test cases to generate new ones. Our approach can leverage the above methods to generate SQL statements.

Differential testing in DBMSs. In DBMS testing, differential testing involves feeding the same SQL statements into multiple DBMSs and observing discrepancies in their execution results. Differential testing is often used to detect logic bugs in DBMSs [20, 28, 50, 62]. Differential testing can also be applied to detect performance bugs in DBMSs [40]. For example, Apollo [40] inputs the same SELECT statement into different versions of the same DBMS to detect performance regression bugs. Our approach executes the same SQL statements on the equivalent databases constructed by different DDL sequences and can be applied to individual DBMSs to test their proprietary syntax features.

Metamorphic testing in DBMSs. In DBMS testing, metamorphic testing involves constructing metamorphic relations between the input and output of SQL statements and detecting any violations of these relations. Some approaches construct databases that return the same query results for SELECT statements [43, 52]. For example, Mozi [43] and Radar [52] construct databases with different configurations and data constraints, respectively, for the same SELECT statement. Other approaches construct equivalent SELECT statements on the same database [14, 15, 27, 31, 41, 46–48, 56, 61]. For instance, DQP [15] adds various query hints to a given SELECT statement, exploring equivalent query executions on the same database. Researchers further construct metamorphic relations for transaction bug detection [20–22, 24, 35, 39] and graph database system bug detection [33, 55, 59–62, 64]. Our approach is a general method that constructs equivalent databases with the same database schema, where the DBMS is expected to return identical execution results for identical SQL statements.

6 CONCLUSION

Relational DBMSs manage user-visible database schemas and various schema-related information for storing and managing database structures. In this paper, we propose DDLCheck, a novel and general approach to automatically detect schema-related logic bugs in relational DBMSs. DDLCheck constructs equivalent database schemas through different DDL sequences and then compares execution results across these sequences for the same SQL statements. We implement and evaluate DDLCheck on six widely-used relational DBMSs, and have detected 34 bugs, of which 29 have been confirmed as new, and 9 have been fixed by DBMS developers.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 2024. CockroachDB Homepage. https://www.cockroachlabs.com/.
[2] 2024. DB-Engines Ranking. https://db-engines.com/en/ranking.
[3] 2024. Failed to INSERT a proper value when no FOREIGN KEY violation. https://jira.mariadb.org/browse/MDEV-34105.
[4] 2024. Inconsistent REPLACE behaviors. https://jira.mariadb.org/browse/MDEV-35115.
[5] 2024. MariaDB Homepage. https://mariadb.org/.
[6] 2024. MySQL Customers by Industry. https://www.mysql.com/customers/.
[7] 2024. MySQL Homepage. https://www.mysql.com.
[8] 2024. PostgreSQL Homepage. https://www.postgresql.org/.
[9] 2024. SQLancer Homepage. https://github.com/sqlancer/sqlancer.
[10] 2024. SQLite Homepage. https://www.sqlite.org/index.html.
[11] 2024. SQLsmith. https://github.com/anse1/sqlsmith.
[12] 2024. TiDB Homepage. https://www.pingcap.com/?from=en.
[13] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O. Laleye, and Sarfraz Khurshid. 2008. Query-Aware Test Generation Using a Relational Constraint Solver. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 238–247.
[14] Jinsheng Ba and Manuel Rigger. 2024. CERT: Finding Performance Issues in Database Systems Through the Lens of Cardinality Estimation. In *Proceedings of International Conference on Software Engineering (ICSE)*. Article 133, 13 pages.
[15] Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proceedings of International Conference on Management of Data (SIGMOD)* (jun 2024).
[16] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. 2007. QAGen: Generating Query-Aware Test Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 341–352.
[17] Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible Database Generators. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 1097–1107.
[18] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A Structured English Query Language. In *Proceedings of ACM SIGFIDET Workshop on Data Description, Access and Control (SIGFIDET)*. 249–264.
[19] Edgar F Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (jun 1970), 377–387.
[20] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. 2022. Differentially Testing Database Transactions for Fun and Profit. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Article 35, 12 pages.
[21] Ziyu Cui, Wensheng Dou, Yu Gao, Dong Wang, Jiansen Song, Yingying Zheng, Tao Wang, Rui Yang, Kang Xu, Yixin Hu, Jun Wei, and Tao Huang. 2024. Understanding Transaction Bugs in Database Systems. In *Proceedings of International Conference on Software Engineering (ICSE)*. Article 163, 13 pages.
[22] Ziyu Cui, Wensheng Dou Dou, Yu Gao, Rui Yang, Yingying Zheng, Jiansen Song, Yuan Feng, and Jun Wei. 2025. Simple Testing Can Expose Most Critical Transaction Bugs: Understanding and Detecting Write-Specific Serializability Violations in Database Systems. *Proceedings of the VLDB Endowment (PVLDB)* 18 (2025).
[23] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. 2013. Automating the database schema evolution process. *The VLDB Journal (VLDBJ)* 22, 1 (Feb. 2013), 73–98.
[24] Wensheng Dou, Ziyu Cui, Qianwang Dai, Jiansen Song, Dong Wang, Yu Gao, Wei Wang, Jun Wei, Lei Chen, Hanmo Wang, Hua Zhong, and Tao Huang. 2023. Detecting Isolation Bugs via Transaction Oracle Construction. In *Proceedings of International Conference on Software Engineering (ICSE)*. 1123–1135.
[25] Jingzhou Fu, Jie Liang, Zhiyong Wu, and Yu Jiang. 2024. Sedar: Obtaining High-Quality Seeds for DBMS Fuzzing via Cross-DBMS SQL Transfer. In *Proceedings of International Conference on Software Engineering (ICSE)*. Article 146, 12 pages.
[26] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2023. Griffin: Grammar-Free DBMS Fuzzing. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Article 49, 12 pages.
[27] Ying Fu, Zhiyong Wu, Yuanliang Zhang, Jie Liang, Jingzhou Fu, Yu Jiang, Shanshan Li, and Xiangke Liao. 2025. THANOS: DBMS Bug Detection via Storage Engine Rotation Based Differential Testing. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 12.
[28] Bogdan Ghit, Nicolas Poggi, Josh Rosen, Reynold Xin, and Peter Boncz. 2020. SparkFuzz: searching correctness regressions in modern query engines. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*. Article 1, 6 pages.
[29] Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. 2001. Exploiting Constraint-like Data Characterizations in Query Optimization. In *Proceedings of International Conference on Management of Data (SIGMOD)*, Vol. 30. 582–592.
[30] Zhongxian Gu, Mohamed A. Soliman, and Florian M. Waas. 2012. Testing the accuracy of query optimizers. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*. Article 11, 6 pages.
[31] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*. 345–358.
[32] Kenneth Houkjær, Kristian Torp, and Rico Wind. 2006. Simple and Realistic Data Generation. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 1243–1246.
[33] Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, Wenpin Jiao, and Tao Xie. 2023. GDsmith: Detecting Bugs in Cypher Graph Database Engines. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 163–174.
[34] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proceedings of the VLDB Endowment (PVLDB)* 13, 12 (aug 2020), 3072–3084.
[35] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David Basin, Haixiang Li, and Anqun Pan. 2023. Efficient Black-Box Checking of Snapshot Isolation in Databases. *Proceedings of the VLDB Endowment (PVLDB)* 16, 6 (feb 2023), 1264–1276.
[36] Ioana Ileana, Bogdan Cautis, Alin Deutsch, and Yannis Katsis. 2014. Complete yet Practical Search for Minimal Query Reformulations under Constraints. In *Proceedings of International Conference on Management of Data (SIGMOD)*. 1015–1026.
[37] Anant Jhingran and Pratap Khedkar. 1992. Analysis of recovery in a database system using a write-ahead log protocol. 21, 2 (1992), 175–184.
[38] Zuming Jiang, Jiaju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In *Proceedings of USENIX Security Symposium (USENIX Security)*. Article 277, 17 pages.
[39] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 397–417.
[40] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *Proceedings of the VLDB Endowment (PVLDB)* 13, 1 (sep 2019), 57–70.
[41] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. 2023. Testing Graph Database Engines via Query Partitioning. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 140–149.
[42] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proceedings of the VLDB Endowment (PVLDB)* 14, 3 (nov 2020), 268–280.
[43] Jie Liang, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Chengnian Sun, and Yu Jiang. 2024. Mozi: Discovering DBMS Bugs via Configuration-Based Equivalent Transformation. In *Proceedings of International Conference on Software Engineering (ICSE)*. Article 135, 12 pages.
[44] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *Proceedings of USENIX Security Symposium (USENIX Security)*. 4309–4326.
[45] Shuang Liu, Chenglin Tian, Jun Sun, Ruifeng Wang, Wei Lu, Yongxin Zhao, Yinxing Xue, Junjie Wang, and Xiaoyong Du. 2025. Semantic Conformance Testing of Relational DBMS. *Proceedings of the VLDB Endowment (PVLDB)* (2025).
[46] Xinyu Liu, Qi Zhou, Joy Arulrai, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries. In *Proceedings of International Conference on Software Engineering (ICSE)*. 225–236.
[47] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1140–1152.
[48] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Vol. 4. Article 211, 30 pages.
[49] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Article 38, 16 pages.
[50] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 618–622.
[51] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing Database Systems via Differential Query Execution. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 2072–2084.
[52] Jiansen Song, Wensheng Dou, Yu Gao, Ziyu Cui, Yingying Zheng, Dong Wang, Wei Wang, Jun Wei, and Tao Huang. 2024. Detecting Metadata-Related Logic Bugs in Database Systems via Raw Database Construction. *Proceedings of the*

VLDB Endowment (PVLDB) 17, 8 (may 2024), 1884–1897.

[53] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1493–1509.

[54] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. COBRA: making transactional key-value stores verifiably serializable. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Article 4, 18 pages.

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

[56] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. 1, 1, Article 55 (2023), 26 pages.

[57] Manasi Vartak, Venkatesh Raghavan, and Elke A. Rundensteiner. 2010. QRelX: Generating Meaningful Queries That Provide Cardinality Assurance. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1215–1218.

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

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

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

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

[62] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. 2022. Finding Bugs in Gremlin-Based Graph Database Systems via Randomized Differential Testing. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 302–313.

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

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