EPSC: Testing Database Management Systems via Equivalent Prepared Statement Construction

CHI ZHANG, Tsinghua University, China
JIE LIANG, Beihang University, China
ZHIYONG WU, Tsinghua University, China
DALONG SHI, AVIC International Digital Network Technology Co., Ltd., China
LINZHANG WANG, Nanjing University, China
YU JIANG*, Tsinghua University, China

Database Management Systems (DBMSs) serve as the backbone for efficient data access and modification through the Structured Query Language (SQL). Bugs in DBMSs may prevent otherwise valid statements from executing or even cause statements to return incorrect results. Prior research has primarily focused on detecting bugs in ordinary SQL statements. In contrast, prepared statements—a language feature widely used in production environments to improve the performance of repeated queries and to guard against SQL injection—have received far less attention, and the potential bugs within them remain insufficiently explored.

In this paper, we present a general black-box approach, termed *Equivalent Prepared Statement Construction (EPSC)*, to detect logic bugs in both ordinary and prepared SQL statements. The key insight of EPSC is that Data Manipulation Language (DML) and Data Query Language (DQL) statements can be executed in two equivalent forms—ordinary statements and prepared statements—which should exhibit consistent behavior and produce identical results. For instance, a `SELECT` statement can be transformed into its prepared statement form by extracting literal values as bound parameters; both forms are expected to yield the same output. Any inconsistency between them indicates the presence of a bug in the target DBMS. To evaluate the effectiveness of EPSC, we applied it to seven mature DBMSs: MySQL, MariaDB, TiDB, PostgreSQL, CockroachDB, SQLite3, and DuckDB. In total, EPSC uncovered 52 unique bugs, of which 50 have been confirmed and 13 have already been fixed. Moreover, our experimental results demonstrate that EPSC effectively detects logic bugs that existing approaches fail to identify. We believe that the simplicity and broad applicability of EPSC can significantly enhance the reliability of DBMS implementations.

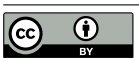
CCS Concepts: • **Information systems → Database query processing**; • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Prepared statement, logic bug, metamorphic testing

ACM Reference Format:
Chi Zhang, Jie Liang, Zhiyong Wu, Dalong Shi, Linzhang Wang, and Yu Jiang. 2026. EPSC: Testing Database Management Systems via Equivalent Prepared Statement Construction. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 176 (June 2026), 25 pages. https://doi.org/10.1145/3802053

*Yu Jiang is the corresponding author.

Authors' Contact Information: Chi Zhang, chi-zhang@mail.tsinghua.edu.cn, Tsinghua University, China; Jie Liang, liangjie.mailbox.cn@gmail.com, Beihang University, China; Zhiyong Wu, 253540651@qq.com, Tsinghua University, China; Dalong Shi, shidl@avic.com, AVIC International Digital Network Technology Co., Ltd., China; Linzhang Wang, lzwang@nju.edu.cn, Nanjing University, China; Yu Jiang, jiangyu198964@126.com, Tsinghua University, China.



ACM 2836-6573/2026/6-ART176
https://doi.org/10.1145/3802053

1 Introduction

Database Management Systems (DBMSs) serve as critical software infrastructure, storing and managing data for a wide range of essential applications. Structured Query Language (SQL) statements serve as inputs to DBMSs, enabling users to specify the operations to be performed on data. DBMSs parse SQL statements, choose an appropriate execution plan, perform the corresponding operations, and return the results. However, given the complexity of DBMS implementations, bugs are likely to arise during the parsing, optimization, and execution of SQL statements. These bugs can lead to system crashes, produce unexpected errors, or more subtly, to logic bugs [25], which cause the DBMS to silently produce incorrect results, making them hard for developers and users to notice.

Recently, various approaches have been proposed for detecting logic bugs in DBMSs. Logic bugs, unlike other types of bugs, seldom cause noticeable system anomalies, making them difficult to detect. Consequently, detecting logic bugs has become a major focus in DBMS testing research. Most existing approaches rely on metamorphic testing to construct effective test oracles. Metamorphic relations form the core of metamorphic testing. Existing approaches leverage a variety of equivalence relations in DBMSs as metamorphic relations to determine whether logic bugs are triggered, including, but not limited to, equivalence across different systems [29], clauses [25, 31], expressions [15, 26, 35], and configurations [2, 17]. While these methods have successfully detected a large number of logic bugs in widely used DBMSs, they have concentrated exclusively on bugs in ordinary statements, neglecting another critical language feature—prepared statements.

Prepared statements were introduced as a specialized language feature to enhance the efficiency of repeated executions, as production environments commonly involve numerous statements that are nearly identical except for their literal values. In contrast to ordinary statements—which are executed directly—prepared statements allow literals to be specified as parameters. The DBMS parses, compiles, and optimizes the prepared statement once, and for multiple bound parameters, it can avoid redundant compilation and optimization, thereby improving efficiency. Because the parameters of prepared statements are undetermined during compilation and optimization, multiple optimizations applicable to ordinary statements cannot be employed. For instance, a prepared statement may produce several sub-optimal query plans, with one selected only at execution time. Moreover, as prepared statements play a key role in preventing SQL injection attacks [4], the bound parameters must be parsed separately. These processes cause the execution paths of prepared statements to differ from those of their equivalent ordinary counterparts. Consequently, approaches that examine only ordinary statements are unable to uncover bugs specific to prepared statements, and it remains unclear how existing approaches could detect bugs that may arise during the compilation or separate execution phases of prepared statements. Nevertheless, the equivalence between the two forms provides an opportunity to establish a metamorphic relation that serves as an effective test oracle.

In this paper, we present *Equivalent Prepared Statement Construction (EPSC)*, a general approach for uncovering logic bugs in the implementation of both prepared and ordinary forms of DML and DQL statements in DBMSs. Our key insight is that prepared statements may follow different execution paths in the DBMS during compilation, optimization, and execution, compared with their equivalent ordinary statements; this divergence can be exploited to detect logic bugs. While prior logic-bug detection techniques primarily rely on semantic equivalence via query rewriting over ordinary statements, EPSC exploits the implementation divergence introduced by prepared statements, which has not been systematically explored before. The high-level idea of EPSC is to contrast the results of a prepared statement with those of an equivalent ordinary statement. Any inconsistency can indicate the presence of a logic bug. Formally, let D be a database and P a prepared statement. Executing P with parameters R in D yields the result $P(D, R)$. For the

corresponding ordinary statement O_{P+R}, which replaces the parameter placeholders in the prepared statement with R, if $P(D, R) \neq O_{P+R}(D)$, we regard this as evidence of a logic bug.

Listing 1. An illustrative example caused by incorrect type hint handling in CockroachDB.

```
-- The ordinary statement
SELECT IF(FALSE, NULL, 4) IS NOT NAN; -- {TRUE} ✔

-- The prepared statement
PREPARE ps(BOOL, UNKNOWN, INT) AS SELECT IF($1, $2, $3) IS NOT NAN;
EXECUTE ps(FALSE, NULL, 4); -- {NULL} 🐞
```

Listing 1 shows an illustrative example EPSC found in CockroachDB, which was caused by incorrect handling of type hints. In the ordinary statement, since the first argument of the `IF` operator is `FALSE`, the operator should evaluate its third argument and return 4. The query then returns `TRUE` because 4 is not `NaN`. Then, we construct the equivalent prepared statement by extracting the literal values `FALSE`, `NULL`, and 4 as parameters, replacing them with the placeholders "`$1`", "`$2`", and "`$3`", and defining their types as `BOOL`, `UNKNOWN`, and `INT`, respectively. The `NULL` value has no inherent type, whereas `FALSE` and 4 are originally of types `BOOL` and `INT`, respectively. However, when the equivalent prepared statement is executed, the query incorrectly returns `NULL`. According to feedback from CockroachDB developers, this occurred because the type hint for `NULL` was set to `UNKNOWN` when defining the prepared statement. This caused the other two arguments of the `IF` operator to be interpreted with incorrect types, ultimately producing an erroneous result. This issue can affect multiple binary expressions, including `COALESCE` and `CASE WHEN`. Since the affected binary expressions consistently produce incorrect results due to a feature specific to prepared statements (i.e., type hints), existing test oracles based on equivalence transformations of `SELECT` statements cannot detect this issue when applied to the `SELECT` within the prepared statement. In contrast, EPSC exploits the divergent parts of the execution paths between equivalent ordinary and prepared statements to expose this logic bug hidden in the prepared statement path.

By following execution paths distinct from those of ordinary statements, prepared statements can reveal logic bugs when the two statements produce different results. Furthermore, we observed that DBMSs exhibit another type of logic bug, in which a statement that should have triggered an error is executed and produces a result. Detecting this type of logic bug is challenging. Fuzzing typically generates a large number of normal and abnormal test cases and feeds them into the DBMS under test, determining the presence of bugs based on the system's behavior or output. Due to the large number of abnormal test cases generated and the repeated triggering of certain common errors, existing testing methods, such as SQLancer, ignore these errors and treat them as normal behavior.[1] However, the erroneous parts may exhibit inconsistent behavior between equivalent ordinary and prepared statements. In DBMSs, certain optimizations, such as short-circuit evaluation, may cause parts of a statement's expressions to be skipped during execution. If an erroneous part is skipped in an ordinary statement but not in a prepared statement—because the bound parameters are unknown and the optimization cannot be applied—the two equivalent statements may exhibit inconsistent behavior, with one executing successfully and returning a result while the other triggers an error. This inconsistency requires us to determine whether the correctly executed statement skipped the erroneous part or whether the erroneous part was executed incorrectly, producing an incorrect result—that is, a logic bug. Treating inconsistent behavior in equivalent statements as a logic bug in the correctly executed statement without careful analysis can lead to a lot of false positives.

[1] https://github.com/sqlancer/sqlancer/blob/main/src/sqlancer/common/query/ExpectedErrors.java

To mitigate false positives arising from the skipping of erroneous parts in statements due to DBMS optimizations—leading to behavior inconsistencies—we introduce an error validation mechanism. Specifically, we traverse and force the evaluation of each expression in the statement without error. If an expression produces the same error as in the other statement, this indicates that the erroneous part was skipped, and the error is considered expected. Otherwise, the correctly executed statement may have executed the erroneous part incorrectly, in which case our method generates an alarm.

We implemented our approach as a practical DBMS testing tool and evaluated it on 7 widely-used and extensively-tested DBMSs, including MySQL, MariaDB, TiDB, PostgreSQL, CockroachDB, SQLite3, and DuckDB. Overall, EPSC identified 52 unique, previously unknown bugs, comprising 28 logic bugs and 24 unexpected errors. Among these, 13 have been fixed, 50 have been confirmed, and 2 remain under analysis by developers. In summary, we make the following contributions:

- Prepared statements are widely used in production environments, yet existing DBMS testing methods do not support them. We identify that the differences between prepared and ordinary statements can be exploited to construct effective test oracles.
- We introduce EPSC, a black-box method for testing DBMSs by constructing prepared statements equivalent to ordinary statements; any discrepancies between them indicate bugs.
- We implemented the proposed method and assessed its effectiveness on 7 widely used, well-tested DBMSs, detecting a total of 52 unique bugs. Finally, we performed a comprehensive comparison with state-of-the-art techniques.

2 Background

Prepared statements. Prepared statements, also known as parameterized statements, were designed to accommodate the frequent occurrence of nearly identical statements in production environments, differing only in certain literal values. In most DBMSs, such statements are defined using the `PREPARE` keyword. The varying literals are treated as parameters, typically represented by symbols such as `?` or `$n`. This allows the statement to be parsed, compiled, and optimized only once. For subsequent executions with different literal values, it is sufficient to bind the parameters and run the statement using `EXECUTE` statements, thereby reducing compilation and optimization overhead and improving execution performance.

Since the bound parameters in prepared statements are unknown during compilation, many optimizations that apply to their equivalent ordinary statements, such as short-circuit evaluation and constant folding [24], cannot be performed. Prepared statements may use distinct mechanisms for query plan generation [12] and selection [6]. Additionally, during execution, variables must be parsed in the `EXECUTE` statement, and SQL injection prevention must be enforced. Consequently, prepared statements may follow execution paths that differ from those of ordinary statements throughout parsing, compilation, optimization, and execution.

DBMS logic bugs. DBMSs, due to their complex implementations and optimizations, are prone to various bugs. Logic bugs [25] are among the most common types of bugs in DBMSs and cause the system to produce incorrect results silently for certain statements. Because their occurrence does not produce obvious system behavior, they are difficult for developers and users to detect. These bugs are critical for DBMSs, as they can occur in statements such as `SELECT`, leading users to obtain incorrect results, or in statements such as `INSERT`, `DELETE`, and `UPDATE`, which may construct incorrect database states and consequently affect all subsequent queries. Furthermore, such bugs may cause inconsistencies between prepared and ordinary statements (e.g., the ordinary statement produces the correct result, whereas the prepared one does not). This can mislead developers who use ordinary statements for debugging during development, while the deployed applications execute equivalent prepared statements, resulting in inconsistent outcomes that propagate to the

upper-layer applications. In this paper, we broaden the scope of logic bugs to include both incorrect results on valid statements and the production of results on statements that should trigger errors. We focus on logic bugs that occur in the ordinary and prepared forms of DML and DQL statements.

Metamorphic testing. Logic bugs that result in incorrect outputs often lack observable system-level symptoms—such as crashes, exceptions, or hang—making them hard to detect for both users and developers. Metamorphic testing [7] has been introduced to construct test oracles for detecting such bugs. Metamorphic testing generates follow-up test cases from existing ones and their results, with the expectation that the outcomes of the follow-up cases should conform to specific metamorphic relations defined with respect to the original cases. Any violation of these relations indicates the presence of logic bugs. EPSC, the method proposed in this paper, is a metamorphic testing approach. It randomly generates ordinary statements and constructs corresponding equivalent prepared statements by replacing one or more literal values with bound parameters, expecting both statements to produce identical results. Any discrepancy reveals a logic bug. Because ordinary and prepared statements (along with their corresponding `EXECUTE` statements) follow different paths during parsing, compilation, optimization, and execution, this approach can detect logic bugs that occur along the buggy path of either the ordinary or the prepared statement.

Short-circuit optimization. Short-circuit optimization, commonly applied in compilers across different programming languages, enables the evaluation of an expression to terminate as soon as its value is determined. For instance, in the expression `TRUE OR 1/0`, SQLite3 returns `TRUE` immediately without evaluating the right-hand operand, regardless of its contents. This approach can significantly enhance DBMS performance, particularly when the optimized-away expressions are complex. Nevertheless, this optimization cannot be applied to prepared statements when the bound parameters are unknown, posing challenges for EPSC in determining whether the inconsistent behavior between equivalent ordinary and prepared statements indicates a logic bug. For example, in the expression `TRUE OR 1/0`, using `TRUE` as a parameter prevents the DBMS from applying short-circuit optimization on the prepared statement, because the value of the left operand is unknown at compile time. As a result, executing the prepared statement triggers a division-by-zero error. Treating the ordinary statement that returns `TRUE` as triggering a logic bug may result in false positives.

3 Approach

In this work, we present EPSC, a lightweight approach for detecting logic bugs in DBMSs, by systematically exploiting the equivalence between prepared statements and their corresponding ordinary statements to expose divergences in query compilation, optimization, and execution. Our core idea is to compare the behavior and results of an ordinary statement with its equivalent prepared statement. Any inconsistency may indicate a logic bug. A *prepared statement* refers to a statement in which some literals are unknown; the statement is first compiled and then executed with bound parameters. We refer to statements that have no parameters and do not separate definition and execution as *ordinary statement*.

3.1 Approach Overview

Figure 1 illustrates our approach. In testing, EPSC maintains two database instances: the original instance executes only ordinary statements, while the reference instance executes both the equivalent prepared statements and the ordinary statements that cannot be executed as prepared statements. The goal of step ① is to construct the database state and to test DML statements using prepared statements. In step ①, we randomly generate a database state by producing a large number of Data Definition Language (DDL), Transaction Control Language (TCL), and DML statements to define

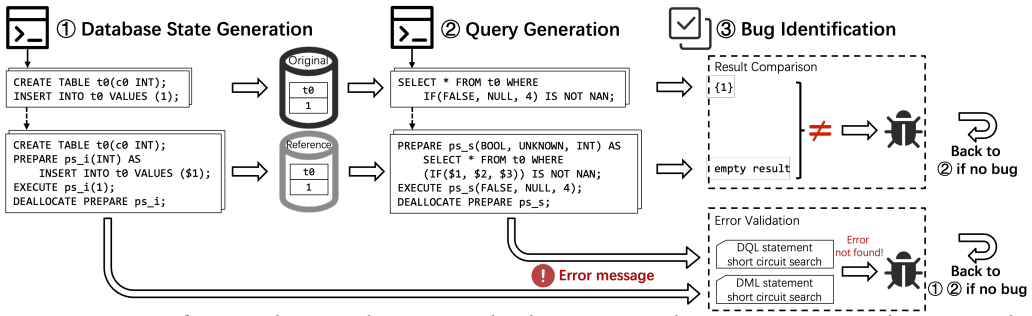


Fig. 1. Overview of EPSC. The core idea is to randomly generate ordinary statements and construct their corresponding prepared statements in steps ① and ②, and then detect logic bugs in step ③.

tables, views, and indexes, and to insert data. During this process, DDL and TCL statements that cannot be converted into prepared statements are executed identically on both the original and reference databases, e.g., the definitions of `t0` in Figure 1. By contrast, most DML statements can be transformed into prepared statements, and we execute these equivalents on the reference database, as illustrated by `ps_i` in Figure 1, which inserts the value 1 into table `t0`.

Step ② tests DQL statements (i.e. SELECT) with prepared statements and exposes inconsistencies in the database state caused by logic bugs in the DML statements executed in step ①. Step ② begins when step ① produces more than a predefined number of statements and the database contains at least one non-empty table, which prevents subsequent queries from trivially returning results due to empty tables and thus improves testing efficiency. In step ②, we generate a single SELECT query based on the current database state. We then execute the query in two forms—the ordinary statement on the original database and the prepared statement on the reference database. The prepared statement is constructed by randomly designating certain literals as parameters.

Step ③ captures and validates any inconsistencies triggered in steps ① and ②. In step ① and ②, when equivalent statements behave inconsistently on the original and reference databases—one raising an error while the other succeeds—we must determine whether this inconsistency stems from a logic bug, or from short-circuit evaluation masking a genuine syntax or semantic error. Step ③ incorporates an error validation mechanism to detect short-circuit optimizations that may have bypassed the error. If no such optimizations are found, the inconsistency is attributed to an issue requiring further analysis. If, however, erroneous parts skipped by optimizations are detected, the error is considered expected. In this case, the process either returns to the previous step to continue generating new statements, or terminates the current testing iteration if the inconsistency causes a divergence between the original and reference database states.

In step ①, if the equivalent statements exhibit consistent behavior—both executing successfully or both triggering errors—we do not immediately proceed to step ③ to determine whether a logic bug exists. This is because logic bugs occurring in step ① may cause inconsistencies between the ordinary and reference database, which will eventually be reflected in the results of the equivalent SELECT statements generated in step ②. If both the ordinary and prepared statements succeed in step ②, we proceed to the result comparison in step ③ to verify whether their outputs match. Any inconsistency may indicate the presence of a logic bug. If the results are consistent, we proceed to step ② to generate additional queries that fully exercise the current database state, followed by step ③, repeating this process until the number of queries reaches a predefined threshold.

3.2 Statement Generation

In steps ① and ②, statements are randomly generated according to the syntax rules of the DBMSs being tested. In step ①, a large number of tables are first randomly generated, and based on

the information of these tables, views, indexes, inserted values, and transactions are randomly constructed. Certain configuration definition are also generated in step ①, some of which are crucial to the prepared statements. For example, in PostgreSQL, the configuration `plan_cache_mode` determines whether a prepared statement is executed using a general query plan or a parameter-specific query plan.[2] As shown in Listing 12, the illustrated bug can only be triggered under the `force_generic_plan` setting of `plan_cache_mode`. During testing, we randomly select values for generated configurations. In step ②, queries are randomly generated based on the database metadata constructed in step ①, incorporating all features supported by the target DBMS.

Equivalent prepared statements are constructed for the DML and DQL statements in steps ① and ②. To construct an equivalent prepared statement, we randomly select literals in the statement and replace them with placeholders supported by the DBMS. If the statement contains no literals, it is executed as an ordinary statement. In this process, the corresponding literal definitions, `EXECUTE`, and `DEALLOCATE` statements are generated following the syntax rules of the target DBMS. In DBMSs with strict type systems, such as PostgreSQL and CockroachDB, parameter types must be explicitly specified in prepared statements, whereas literals in ordinary statements do not require explicit typing. This can lead to a mismatch between the types inferred automatically in ordinary statements and those specified in prepared statements, potentially resulting in different execution outcomes. To eliminate false positives due to potential type mismatches, we perform explicit type casting to literals extracted as parameters in the corresponding ordinary statements. Since the selection of literals used as bound values is random, literals that are not selected are not subjected to explicit type casting, allowing them to be used as mixed-type expressions or schema-dependent literals.

In both prepared statements and ordinary statements, literals typically have the same meaning, representing the constant value they denote. However, there is an exception in `SELECT` statements: a single numeric literal appearing in `GROUP BY` or `ORDER BY` clauses acts as an alias for a column in the result set. Converting such a literal into a parameter for a prepared statement would lead to inconsistencies, because in prepared statements, DBMSs always evaluate the parameterized expression in `GROUP BY` and `ORDER BY` as the bound parameter value rather than the aliased column. Therefore, when selecting literals as parameters, we avoid choosing those that appear alone in `GROUP BY` or `ORDER BY` clauses.

3.3 Error Validation

During testing, when equivalent ordinary and prepared statements exhibit different execution behaviors—specifically, one raises an error while the other executes successfully—we need to analyze whether the error is expected and was skipped in the other statement due to short-circuit optimization, or the successful statement triggers a logic bug. In short, the error validation mechanism traverses the successfully executed statement, extracts every expression and its subexpressions, and forces them to execute with the same clause inputs as in the original statement to check whether the same error is triggered. If so, the error is considered one that was skipped due to short-circuit optimization. If no such error is triggered, we report the inconsistency as a potential bug.

A natural approach to search for the potential erroneous parts is to extract all subexpressions from each clause and iteratively substitute them back into the original clause to check whether any of them reproduce the same error. However, multiple DBMSs, such as PostgreSQL, impose strict type constraints on expressions. For example, a non-boolean expression cannot be directly placed in a `WHERE` clause. As a result, this approach is not applicable to DBMSs with strict type systems. Even in type-lenient systems such as SQLite3, placing arbitrary expressions in clauses may trigger implicit type conversions, potentially leading to different error messages. To address this challenge,

[2]https://www.postgresql.org/docs/current/runtime-config-query.html#GUC-PLAN-CACHE-MODE

Algorithm 1: Detecting skipped erroneous parts caused by short-circuit optimization

```
Input: SuccessfulStatement s,
       ErrorMessage err_msg
Output: IsErrorSkipped res
1  execution_order ← GetClauseAnalysisOrder(s);
2  foreach clause in execution_order do
3      expr_in_clause ← ExtractExpr(clause);
4      sub_expr_list ←
         DecomposeExpr(expr_in_clause);
5      foreach expr in sub_expr_list do
6          q ← QueryConstruction(s, expr);
7          output ← QueryExecution(q);
8          if ErrMsgComparison(output, err_msg)
             then
9              return true;
10         end
11     end
12     if ClauseSpecificCheck(clause) then
13         return true;
14     end
15 end
16 return false;
```

Algorithm 2: Algorithm of QueryConstruction

```
Input  : Statement s, Subexpression expr
Output : Query q
1  type ← GetStatementType(s);
2  clause_type ← GetClauseType(s, expr);
3  if type is SELECT then
4      if clause_type ∈ {LIMIT, OFFSET} then
5          q ← InitializeNewSelectQuery();
6          q.SelectList ← {expr};
7          q.FromClause ← GetNecessaryTables(s);
8      else
9          q ← Copy(s);
10         RemovePreviouslyAnalyzedClauses(q);
11         q.SelectList ← {expr};
12 else if type ∈ {UPDATE, DELETE} then
13     q ← InitializeNewSelectQuery();
14     q.SelectList ← {expr};
15     q.FromClause ← GetNecessaryTables(s);
16 else if type is INSERT then
17     q ← InitializeNewSelectQuery();
18     q.SelectList ← {expr};
19     q.FromClause ← ConstructValuesTable(s);
20 return q;
```

we note that in DBMSs, an expression should produce the same result for the same input regardless of the clause in which it appears. Therefore, we move the evaluation of expressions from various clauses into the SELECT clause and check whether the subexpressions trigger the same error.

Algorithm 1 illustrates our approach, which takes as input the successfully executed statement s and the error message err_msg produced by its equivalent ordinary or prepared statement, and returns whether there are errors that are skipped by short-circuit optimizations. Considering that clauses in a statement execute in a specific order and often receive input from preceding clauses rather than directly from tables, we analyze the clauses in a particular order to preserve the values used for expression evaluation as accurately as possible, avoiding situations where different data might trigger different errors. Specifically, clauses in a SELECT statement are executed in the following order: FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT/OFFSET. We analyze each clause in the reverse of this order. For UPDATE and DELETE statements, we perform a similar reverse-order analysis on the WHERE and other relevant clauses. For INSERT statements, we focus solely on the column- or table-level check constraints. The analysis order of the clauses can be obtained using the GetClauseAnalysisOrder function, shown in line 1. Therefore, we analyze the expressions in each clause in the reverse of this execution order, as illustrated by the foreach loop in line 2.

We then analyze each subexpression individually, as the foreach loop shown in line 5. For each subexpression and the clause in which it appears, we construct a query q with QueryConstruction. This step varies depending on the type of statement and clause. We illustrate this process using Algorithm 2. First, for SELECT statements, the process depends on the clause type. If the subexpression resides in a LIMIT or OFFSET clause, we generate a new query q by placing the subexpression in the SELECT list and including only the strictly necessary tables in the FROM clause. For subexpressions in any other clauses of a SELECT statement, we derive q from the original statement s by removing

any clauses that have already been analyzed (to enforce the execution order) and replacing the original SELECT list with the extracted subexpression. Second, for UPDATE and DELETE statements, we construct a standard SELECT query q. We place the extracted subexpression in the SELECT clause and set the table being manipulated as the FROM clause. Finally, for INSERT statements, we focus on validating column or table constraints. We construct a SELECT query where the extracted subexpression is placed in the SELECT clause. The FROM clause is populated by a temporary table structure identical to the target table, containing exclusively the values intended for insertion.

Listing 2. Example of error validation.

```
CREATE TABLE t0(c0 INT);
CREATE TABLE t1(c0 INT);
INSERT INTO t0 VALUES (0);
SELECT * FROM t0 LEFT JOIN t1 ON false WHERE (10/t0.c0) OR TRUE; -- {0|null}
PREPARE ps(BOOLEAN) AS SELECT * FROM t0 LEFT JOIN t1 ON FALSE WHERE (10/t0.c0) OR ?;
EXECUTE ps(TRUE); -- Error: division by zero

-- The process of checking the WHERE condition
SELECT (10/t0.c0) OR TRUE FROM t0 LEFT JOIN t1 ON FALSE; -- {TRUE}
SELECT 10/t0.c0 FROM t0 LEFT JOIN t1 ON FALSE; -- Error: division by zero
```

After the constructed query q is executed (i.e., line 7), we check whether it triggers an error. If an error occurs, we compare its message with *err_msg*; if they match, we conclude that the original query indeed contains an erroneous part skipped due to short-circuit optimization, and return *true*. Listing 2 shows the process of checking the WHERE clause, where the same error is triggered when checking the subexpression of the condition.

Listing 3. The error requiring specific checking.

```
SELECT (false IN c0) FROM v0 GROUP BY (false IN c0); -- no error
PREPARE ps AS SELECT (? IN c0) FROM v0 GROUP BY (false IN c0);
    -- column "c0" must appear in the GROUP BY clause or must be part of an aggregate function.
```

Additionally, we observed that some expressions execute successfully in the SELECT clause but trigger errors when placed in specific clauses. Listing 3 presents such an example in DuckDB. The expression `false IN c0` is valid when used simultaneously in SELECT and GROUP BY; however, replacing it with a bound value, i.e., `? IN c0`, renders the GROUP BY clause invalid because it now references a non-grouped column. Although we extracted the expression `false IN c0`, the same error could not be reproduced, as it is inherently tied to GROUP BY semantics. Consequently, we hard-code this specific error message to ensure it is properly filtered. We use the function ClauseSpecificCheck to determine whether such cases exist. Since these cases are relatively rare, we record the corresponding error messages and their characteristics, and then use static analysis of expressions within clauses to check whether they match these recorded error patterns.

If none of the expressions or subexpressions trigger the same error, and the error is not clause-specific, we conclude that no errors were skipped due to short-circuit optimization. Consequently, the equivalent ordinary and prepared statements exhibit inconsistent behavior, and a bug report is generated. In Algorithm 1, for DML statements, we only check whether the WHERE clause and other possible clauses may trigger the same errors, without inspecting the data manipulation process itself. We consider any inconsistency arising during data manipulation to be a definite bug.

Listing 4. An unexpected error found in DuckDB caused by inconsistent numeric parsing.

```
-- The ordinary statement
SELECT - -9223372036854775808; -- {9223372036854775808} ✔

-- The prepared statement
PREPARE ps AS SELECT (- ?);
EXECUTE ps(-9223372036854775808); -- Overflow in negation of integer! 🐞
```

Listing 5. A logic bug found in CockroachDB caused by integer overflow.

```
-- The test case executed in the original database
CREATE TABLE t1 (c0 SMALLINT);
INSERT INTO t1 (c0) VALUES(NULL), (32768); -- ERROR: integer out of range for type int2 ✔

-- The test case executed in the reference database
CREATE TABLE t1 (c0 SMALLINT);
PREPARE ps (unknown, int8) AS INSERT INTO t1 (c0) VALUES($1), ($2);
EXECUTE ps(NULL, 32768);
SELECT * FROM t1; -- {NULL}, {-32768} 🐞
```

Our approach cannot directly distinguish whether the bug is caused by an unexpected syntax or semantic error in a valid statement, or by the unintended execution of a statement containing such errors, as the same type of error may be triggered across different DBMSs for either reason. Listing 4 shows an unexpected error found in DuckDB. The negation of a boundary value triggered an integer overflow error in the prepared statement. This occurred because the `SELECT` and `EXECUTE` statements use different numeric parsing mechanisms. In the ordinary statement, the value `9223372036854775808` was first parsed as an `int128` constant, so subsequent double negations did not cause any issue. In contrast, the `EXECUTE` statement parsed `-9223372036854775808` directly as an `int64` constant—the smallest value representable by this type—and negating it again resulted in an overflow error. Not all errors are unexpected; conversely, a successfully executed statement may trigger a logic bug. Listing 5 illustrates a logic bug discovered in CockroachDB triggered by the prepared statement caused by integer overflow. In the ordinary statement, inserting a value exceeding the range of a `SMALLINT` column triggers an integer overflow error, whereas the equivalent prepared statement produces no error. Manual analysis revealed that the prepared statement silently caused an integer overflow: the attempt to insert `32768` should have triggered an overflow error, but the value `-32768` was actually inserted.

If one of the equivalent ordinary and prepared DML statements successfully performs data manipulation while the other fails, we terminate the current test iteration and proceed to the next one. This is necessary because such inconsistencies lead to discrepancies between the original and reference database instances, causing all queries generated in subsequent step ② to produce inconsistent results and potentially resulting in false positives.

3.4 Result Comparison

Both the DML statements generated in step ① and the DQL statements generated in step ② may trigger logic bugs, we adopt a unified approach to detect their manifestation. A naive way to detect bugs in DML statements is to immediately compare the states of the original and reference databases after each DML execution in step ①; any mismatch would indicate that the most recently executed DML statement triggered a logic bug. While this allows instant detection of DML-induced bugs, our evaluations show that frequent state comparisons incur substantial overhead, reducing testing

performance by 4% (on SQLite3)-22% (on PostgreSQL). Therefore, we check the result consistency of DML statements by comparing the results of the SELECT queries generated in step ②, which simultaneously exposes inconsistencies triggered by these queries.

Listing 6 shows a real bug found in TiDB, illustrating how logic bugs in DML statements can be detected using SELECT queries. In this example, the prepared DELETE statement silently produces an incorrect database state (i.e., it deletes all tuples in t0 despite a false condition), and this inconsistency is exposed by a simple SELECT query. Exposing logic bugs triggered by the DML statements in step ① via the SELECT queries generated in step ② does not introduce false negatives. Because we randomly generate up to 100,000 SELECT queries in step ②, these queries sufficiently explore the database state to reveal any existing inconsistencies.

Listing 6. The logic errors triggered by DML statements can be exposed by SELECT statements.

```
CREATE TABLE t0(c0 FLOAT);
REPLACE INTO t0(c0) VALUES (100000030);

-- The statements executed in the original database
DELETE FROM t0 WHERE t0.c0 NOT LIKE (CASE WHEN false THEN false ELSE t0.c0 END);
SELECT * FROM t0; -- {100000030} ✔

-- The statements executed in the reference database
SET @c = false;
PREPARE prepare_query FROM 'DELETE FROM t0 WHERE t0.c0 NOT LIKE(CASE WHEN false THEN ? ELSE t0.c0
      END)';
EXECUTE prepare_query USING @c;
SELECT * FROM t0; -- empty result 🐞
```

4 Evaluation

We implemented EPSC and evaluated its effectiveness through the following research questions:

- **Q1:** How effective is EPSC at discovering new bugs?
- **Q2:** How effective and efficient is EPSC compared with other test oracles?
- **Q3:** How does the error validation component of EPSC affect false positives and testing performance?

Implementation. EPSC is implemented in 40K lines of Java code, providing a framework for testing DBMSs using prepared statements. We reused the statement generators from SQLancer [27] for statement generation, which are manually written and rule-based. In addition, we extend the generator to support richer JOIN constructions, including self-joins and joins with subqueries. We further add support for stored procedures in PostgreSQL, MySQL, and MariaDB, as well as prepared transactions in PostgreSQL. For generating prepared statements, we first filter out statements that can be executed as prepared statements. We then traverse their ASTs, randomly select literals, and replace them with placeholders supported by the target DBMS. Corresponding EXECUTE statements are also generated to run the prepared statements. Although DBMSs support various types of connectors, each implementing its own interface for prepared statements—for example, the PreparedStatement class in JDBC[3]—these connectors typically translate prepared statements into the native syntax supported by the underlying DBMS. Therefore, we used the prepared statement syntax natively supported by each DBMS whenever possible. The tool and all evaluation results are open-sourced in https://figshare.com/s/69ac376aa97ed00f73d8.

[3]https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html

Experimental setup. All evaluation were conducted on a server equipped with a 128-core AMD EPYC 7742 CPU running at 2.25 GHz and 250 GB of memory, operating on Ubuntu 20.04. All tools were executed under the same configuration with 10 threads to minimize thread-level variance. Comparisons were run for 24 hours, following established fuzzing methodology [16].

Bug report reduction and deduplication. EPSC's test cases include: several thousand DDL/DML statements that create database state and one SELECT query that exposes the inconsistency or triggers an error. To minimize error-inducing test cases, we use a delta-debugging–based automated reducer to shrink them as much as possible. For logic bug test cases, we manually reduce them. First, we shrink the `SELECT` statements by removing identical features from both the ordinary and prepared statements while preserving their result inconsistency. Next, we remove DML and DDL statements from both test cases, again ensuring that the `SELECT` results remain inconsistent. In our evaluation, the manual effort required to minimize a single test case was less than 20 minutes, with most cases completed within just a few minutes. Before submitting bug reports, we deduplicate them by cross-checking the vendors' official issue trackers. For errors, we compare the error messages to identify duplicates. For logic bugs, we check whether the reports share the same language features and exhibit the same behavior across DBMSs.

4.1 Effectiveness

Methodology. We evaluated the effectiveness of EPSC by testing seven widely-used and mature DBMSs, including MySQL [22], MariaDB [19], TiDB [14], PostgreSQL [21], CockroachDB [32], SQLite3 [11], and DuckDB [23]. We selected these DBMSs for two main reasons. First, they are highly popular and mature, and have been extensively tested by existing methods. Among these DBMSs, two rank in the top 10 (i,e,. MySQL and PostgreSQL) and four in the top 20 (i.e., SQLite3 and MariaDB) in terms of overall DBMS popularity according to the DB-Engines Ranking[4], with all seven ranking within the top 100. Moreover, all these DBMSs have been tested by existing works on detecting logic bugs in DBMSs [2, 25–27, 31, 35]. The high popularity and extensive testing history of these DBMSs indicate a lower likelihood of undiscovered bugs, which demonstrates the effectiveness of EPSC. Second, they represent different types of DBMSs, demonstrating that EPSC can discover bugs across diverse DBMS architectures. MySQL, MariaDB, and PostgreSQL represent traditional client–server relational databases, CockroachDB and TiDB represent distributed relational databases, while SQLite3 and DuckDB represent embedded relational databases. We selected the latest release version of each DBMS as the testing target, since newer releases tend to be more stable and robust. In particular, we tested MySQL 9.4.0, MariaDB 12.2.0, TiDB 8.5.3, PostgreSQL 18.0, CockroachDB 25.4.0, SQLite3 3.50.4, and DuckDB 1.4.0. We intermittently ran EPSC over a period of three months, during which we were also developing it. This is a standard practice for evaluating the effectiveness of logic bug detection methods [25, 26, 35].

Results. Table 1 summarizes the number and status of bugs found by EPSC. In total, EPSC discovered 52 previously unknown bugs, including 28 logic bugs and 24 unexpected errors. Among these 52 bugs, 50 have been confirmed, of which 13 have been fixed, and 2 remain under analysis by the developers. These results are highly encouraging, given that the tested DBMSs have already been extensively tested by multiple prior works.

Bug root causes. Among the 52 bugs detected by EPSC, 40 are found in prepared statements. Including 20 prepared statement logic bugs (8 type-related, 6 involving special characters, 1 in error-handling, 1 JOIN-related, and 4 caused by functions producing incorrect results of unknown origin), and 20 prepared statement errors (8 type-related, 6 special character-related, 2 JDBC-related, and 4

[4]https://db-engines.com/en/ranking

Table 1. EPSC found 52 unique bugs in seven mature DBMSs.

DBMS	Bug type		Bug status		
	Logic bug	Unexpected error	Fixed	Confirmed	Open
MySQL	5	2	0	5	2
MariaDB	3	1	2	4	0
TiDB	12	9	3	21	0
PostgreSQL	1	0	0	1	0
CockroachDB	4	8	4	12	0
SQLite	2	0	2	2	0
DuckDB	1	4	2	5	0
Total	28	24	13	50	2

triggered by language features such as functions, GROUP, or EXPLAIN). Most bugs in prepared statements fall into type inference and special character handling. For type inference, mismatches between the expected and actual values in prepared statements can lead to missing, incorrect, or miscast types. For special character handling, additional requirements (e.g., escaping) in prepared statements can result in incorrect values or internal errors.

Among the 52 bugs detected by EPSC, 12 are found in ordinary statements. Including 8 ordinary statements logic bugs (5 optimization-related—1 short-circuit, 1 subquery, 3 joins—and 3 function-related), and 4 ordinary statements errors (3 type-related and 1 special character-related).

For the statement type of the 52 bugs, 7 bugs originate from DML statements generated in step ①, while the remaining 45 originate from `SELECT` statements.

17 of 24 unexpected errors were syntax or semantic errors triggered by DBMSs on entirely valid statements, detected by the error validation mechanism in step ③, while 7 were internal errors. Internal errors are distinct from unexpected syntax or semantic errors. They indicate that the DBMS has entered an internal state that was not anticipated by its developers. When such errors occur, the system typically emits explicit log messages that instruct users to report the issue. For example, in TiDB, we observed the following log message on an internal error bug: "*baseBuiltinFunc.vecEvalInt() should never be called, please contact the TiDB team for help.*"

Bug importance. Among the confirmed logic bugs in databases such as MySQL, MariaDB, and TiDB, 9 have been classified as Major, Critical, or Serious. Developers of CockroachDB expressed their gratitude multiple times for our bug reports. These responses indicate that our work uncovered several high-impact bugs, and that the developers appreciated and actively addressed them.

4.2 Selected Bugs

In this subsection, we first present the logic bugs found in ordinary statements, then those found in prepared statements, and finally the unexpected errors detected by EPSC.

Logic bugs found in ordinary statements. Listing 7 illustrates a logic bug we discovered in SQLite3, caused by the short-circuit optimization. In this test case, a table `t0` with a single column `c0` was created, but no data was inserted into it. In the ordinary `SELECT` query, the expression `max(c0) AND 0` was optimized to `0` by the short-circuit optimization, since the right operand of the `AND` operator was assumed to evaluate to `FALSE`. Consequently, the query returned an empty result because the table `t0` is empty. However, this result is incorrect. In this example, the aggregate `max(c0)` must be computed before evaluating the `AND` operator. When the table is empty, the aggregate function `max(x)` should return `NULL`, after which the `AND` operation should be executed. The correct evaluation should therefore yield `FALSE`, represented as `0`. In the prepared `SELECT`

Listing 7. A logic bug found in Sqlite3 caused by short circuit optimization.

```
CREATE TABLE t0(c0);

-- The ordinary statement
SELECT max(c0) AND 0 FROM t0; -- empty result 🐞

-- The prepared statement
.parameter set $zero 0
SELECT max(c0) AND $zero FROM t0; -- {0} ✔
```

statement, because the right operand of the AND operator was parameterized and thus its value was unknown at optimization time, the short-circuit optimization could not be applied, leading the query to return the correct result.

Listing 8. A logic bug in MariaDB caused by incorrect handling of the JOIN type.

```
CREATE TABLE t0(c0 INT);
CREATE TABLE t1(c0 INT);
INSERT INTO t0 VALUES (1);
INSERT INTO t1 VALUES (1);

SET @a = FALSE;
PREPARE ps FROM "SELECT t0.c0 FROM t1 LEFT JOIN t0 ON FALSE WHERE (? OR ((t0.c0 IS TRUE) IN (
    FALSE)))";
EXECUTE ps USING @a; -- {NULL} ✔
-- Query plan:
-- id table rows Extra
-- 1  t1    1
-- 1  t0    1    Using where; Using join buffer (flat, BNL join)

SELECT t0.c0 FROM t1 LEFT JOIN t0 ON FALSE WHERE (FALSE OR ((t0.c0 IS TRUE) IN (FALSE))); --
    empty result 🐞
-- Query plan:
-- id table rows Extra
-- 1  NULL  NULL Impossible WHERE
```

Listing 8 illustrates a logic bug found by EPSC in MariaDB. This bug-inducing test case involves the creation of tables `t0` and `t1`, each having a column `c0` that contains the value 1. In the prepared statement, table `t1` is left-joined with `t0` using `FALSE` as the join condition. Consequently, `t0` contains only a single `NULL` value after the join. In the `WHERE` clause, we construct an `OR` operator, with the left operand corresponding to a parameter of the prepared statement and the right operand being an `IN` expression. Given that `t0.c0` contains only one `NULL`, the expression `t0.c0 IS TRUE` evaluates to `FALSE`, whereas the `IN` expression evaluates to `TRUE`. Therefore, regardless of the value bound to the left operand, the `OR` expression evaluates to `TRUE`, and the prepared statement is expected to return a single row. We assigned the value `FALSE` to the parameter of the prepared statement and observed the correct result. However, the ordinary statement, in which the placeholder was replaced with the value `FALSE`, produced an empty result, which was unexpected. By obtaining the query plans for both the prepared and ordinary statements, we observed that these two equivalent queries followed different execution paths. The developer confirmed that the `IS TRUE` expression in the WHERE condition incorrectly transformed the left outer join in the ordinary query into an inner join, while the prepared statement executed correctly.

Listing 9. A logic bug found in TiDB caused the IFNULL function to produce an incorrect result.

```
CREATE TABLE t0(c0 DECIMAL NOT NULL);
INSERT INTO t0 VALUES (-1234567890);

-- The ordinary statement
SELECT IFNULL(t0.c0, 'a') FROM t0; -- {-123456789} 🐞

-- The prepared statement
SET @a = 'a';
PREPARE prepare_query FROM "SELECT IFNULL(t0.c0, ?) FROM t0";
EXECUTE prepare_query USING @a; -- {-1234567890} ✔
```

Listing 9 shows a logic bug we found in TiDB, where the IFNULL function produces an incorrect result. In this test case, we first created a table t0 with a DECIMAL column c0, which has a NOT NULL constraint. We then inserted the value -1234567890 into c0, a 10-digit decimal number that is well within the representable range of DECIMAL. In both the ordinary statement and the prepared statement, we constructed a SELECT query containing a single IFNULL function, which returns the first non-NULL value among its arguments (i.e., the column c0 and the character literal 'a' in this case). We expected the query to return the decimal number -1234567890; however, the ordinary statement incorrectly truncated the last digit and returned -123456789, while the prepared statement produced the correct result. This bug can be reproduced with any 10-digit decimal number and is currently under analysis by the developers.

Listing 10. A logic bug found in MySQL caused by the escape character "\".

```
CREATE TABLE t0(c0 INTEGER);
REPLACE INTO t0 VALUES (1);

-- The ordinary statement
SELECT t0.c0 FROM t0 WHERE '1\'#' AND FALSE; -- empty result ✔

-- The prepared statement
PREPARE ps FROM "SELECT t0.c0 FROM t0 WHERE '1\'#' AND FALSE";
EXECUTE ps; -- {1} 🐞
```

Logic bugs found in prepared statements. Listing 10 shows a logic bug we found in MySQL, which is caused by improper handling of the escape character "\" in prepared statement definitions. In this test case, a table t0 with an INTEGER column c0 was created, and a single value 1 was inserted. In the original SELECT statement, the AND operation had a left operand that was a string containing a single quote, which should have been escaped with a backslash, and a right operand of FALSE. The expression should therefore evaluate to FALSE, and the original statement correctly returned an empty result. In the MySQL family of DBMSs, a prepared statement must be enclosed in double quotes at definition time. However, in this context, a single quote escaped by a backslash is incorrectly treated as an ordinary quote. As a result, the expression '1\'#' is interpreted as '1', with the # commenting out the remainder of the statement. Consequently, the condition of the prepared statement evaluates to TRUE, returning all table contents. We reported this issue to the MySQL developers, who confirmed the bug.

Listing 11 shows a logic bug we found in TiDB where the type cast operator returns an incorrect result. In this test case, a table t0 with a CHAR column c0 was created. In the subsequent SELECT

Listing 11. A logic bug found in TiDB related to type cast.

```
CREATE TABLE t0(c0 CHAR );
INSERT INTO t0 VALUES ('a');

-- The ordinary statement
SELECT CAST((CASE WHEN true THEN -1 ELSE t0.c0 END) AS CHAR) FROM t0; -- {'-1'} ✔

-- The prepared statement
SET @b = true;
PREPARE ps FROM 'SELECT CAST((CASE WHEN ? THEN -1 ELSE t0.c0 END) AS CHAR) FROM t0';
EXECUTE ps USING @b; -- {'-'} 🐞
```

query, the result of a `CASE WHEN` expression was cast to the `CHAR` type, which should yield `-1`. While the ordinary statement correctly returned the string `'-1'`, the prepared `SELECT` statement incorrectly returned the string `'-'`. We removed the `CAST` operator from both queries to verify the result of the `CASE WHEN` expression, and found that both returned the correct result. The TiDB developers confirmed the issue but gave no further analysis of the root cause. In TiDB, we found several functions and operators, such as the `HEX` function, `<=>` operator, and `CASE WHEN` operations that yield incorrect results when executed in the form of prepared statements.

Listing 12. A logic bug found in PostgreSQL caused by improper error handler.

```
-- The test case executed in the original database
CREATE TABLE t0(c0 serial, c1 integer);
INSERT INTO t0(c1) VALUES(1/0); -- ERROR: division by zero
INSERT INTO t0(c1) VALUES(2);
SELECT c0, c1 FROM t0; -- {1, 2} ✔

-- The test case executed in the reference database
SET plan_cache_mode = force_generic_plan;
CREATE TABLE t0(c0 serial, c1 integer);
PREPARE ps (integer, integer) AS INSERT INTO t0(c1) VALUES($1/$2);
EXECUTE ps (1, 0); -- ERROR: division by zero
INSERT INTO t0(c1) VALUES(2);
SELECT c0, c1 FROM t0; -- {2, 2} 🐞
```

We also discovered several bugs in DML statements that can only be triggered through prepared statements. Listing 12 shows a logic bug in PostgreSQL resulting from improper error handling. This test case defines a table `t0` with two columns: `c0` as an auto-incrementing integer and `c1` as an integer. During the first insertion into `t0`, we introduce a semantic error—division by zero. In the ordinary statement, we directly use the expression `1/0`, whereas in the `PREPARE` statement, `1` and `0` are provided as parameters. As a result, both the ordinary statement and the `EXECUTE` statement raise a division by zero error upon execution. Since both statements produce the same error, we skip step ③, the error validation step. In the subsequent query generation of step ②, we generate a query to retrieve data from `t1`, and the result validation reveals that the two test cases return different results. We reported the issue to the PostgreSQL developers, who responded that in the ordinary statement, the division by zero occurs before `c0` is incremented, so its first increment takes place during the second insertion, giving it a value of 1. In the prepared statement, the underlying sequence for `c0` advances before the error and is not rolled back, causing a second increment on the second insertion; thus, `c0` takes the value 2. As this logic bug arises from the error handling

of prepared INSERT statements, it is beyond the reach of NoREC [25] and TLP [26], which target SELECT queries, and also DQE [31], whose detection of INSERT-related logic bugs is limited to WHERE-clause conditions.

Listing 13. A logic bug found in SQLite3 caused by inconsistent encoding.

```
-- The test case executed in the original database
PRAGMA encoding = 'UTF-16';
CREATE TABLE t0 (c0 TEXT CHECK (c0 IS TRUE));
INSERT INTO t0(c0) VALUES (x'310a'); -- CHECK constraint failed: c0 IS TRUE ✔

-- The test case executed in the reference database
PRAGMA encoding = 'UTF-16';
CREATE TABLE t0 (c0 TEXT CHECK (c0 IS TRUE));
.parameter set $1 x'310a'
INSERT INTO t0(c0) VALUES ($1); -- 🐞
SELECT c0, (c0 IS TRUE) FROM t0; -- {1|0}
```

The bugs discussed above were all discovered by observing inconsistencies in SELECT query results (i.e., detected via the result comparison in step ③). However, during error validation, we also encountered cases where statements executed successfully yet exposed a logic bug, while the raised error was actually expected behavior. Listing 13 shows a bug found in SQLite3, where the encoding of BLOB parameters is not consistent with the database's configured encoding. In this test case, we constructed a table `t0` with a column `c0` that has a CHECK constraint (`c0 IS TRUE`). The database encoding was set uniformly to UTF-16. However, the prepared statement did not use the database encoding when parsing the BLOB value, causing the value `x'320e'` to pass the constraint when inserted into `t0.c0`, whereas the ordinary statement triggered a CHECK constraint failed error. Further analysis of `t0` and the evaluation of `c0 IS TRUE` revealed that the value was successfully inserted into `t0` and passed a CHECK that returned FALSE.

Listing 5 shows a logic bug we found in CockroachDB caused by integer overflow. In the ordinary INSERT statement, the insertion of 32768 correctly raised an integer overflow error, since 32768 exceeds the maximum value representable by SMALLINT. However, the prepared INSERT statement executed without error. Further inspection of the reference database revealed that the prepared statement had caused an integer overflow and silently inserted an incorrect value, i.e., -32768.

Listing 14. An unexpected error found in DuckDB.

```
CREATE TABLE t0(c0 INTEGER);
INSERT INTO t0(c0) VALUES (1);

-- The ordinary statement
SELECT LENGTH(NULL) FROM t0 GROUP BY NULL;
    -- No function matches the given name and argument types 'length(INTEGER)' 🐞

-- The prepared statement
PREPARE prepare_query AS SELECT LENGTH(?) FROM t0 GROUP BY NULL;
EXECUTE prepare_query(NULL); -- {NULL} ✔
```

Unexpected syntax or semantic errors. Listing 14 shows an unexpected semantic error we found in DuckDB. In the ordinary statement, this error is triggered when a NULL value is used both in the GROUP BY clause and as the argument of the `length` function. The NULL value is implicitly

assigned an INTEGER type, which causes the error. In contrast, in the prepared statement, the `length` function correctly returns NULL when its argument is NULL. In SQLancer, the expression generator for DuckDB does not strictly validate the return types of generated expressions. As a result, it can easily produce a non-string argument for the `length` function during random expression generation. Consequently, such errors are treated as expected semantic errors and thus ignored. In contrast, our approach revealed this issue by detecting the inconsistent behavior between the ordinary and prepared statements.

Listing 15. An unexpected error occurs in TiDB when a bound parameter is used as the grouping key.

```
CREATE TABLE t0(c0 DECIMAL);
INSERT IGNORE INTO t0 VALUES (1);

-- The ordinary statement
SELECT 2 FROM t0 WHERE t0.c0 GROUP BY 1; -- {2} ✔

-- The prepared statement
SET @a = 2;
PREPARE ps FROM 'SELECT ? FROM t0 WHERE t0.c0 GROUP BY 1';
EXECUTE ps USING @a; -- Unknown column '2' in 'group statement' 🐞
```

Listing 15 shows an unexpected semantic error in TiDB that occurs when a bound parameter is used as the grouping key. In SQL, a constant in the GROUP BY clause can represent the alias of a fetched column in the SELECT clause, and the grouping operation is then performed based on that column. However, when a bound parameter is used in the fetched column, the prepared statement raises an "unknown column" error, whereas the ordinary statement executes successfully.

In summary, EPSC can detect bugs arising from a wide range of root causes, including but not limited to issues in optimizations, JOIN operations, functions, operators, encoding, and error handling, demonstrating the broad coverage of logic bugs revealed by EPSC.

4.3 Compared with Other Approaches

Methodology. To answer the second question—how the effectiveness and efficiency of EPSC compare with existing approaches—we executed the tools on SQLite3 3.30.0 for 24 hours, gathering data on the number of unique bugs triggered by each tool, bugs detected exclusively by a given tool, code coverage, and the number of unique query plans. We selected an older version of SQLite3 because multiple bugs had been reported and fixed in this version by previous works. This allows us to determine whether two bug reports correspond to the same underlying bug by checking their respective fixing commits. In this experiment, we identified the fixing commit of each bug using `bisect`, a binary search tool supported by multiple source control systems.

Baselines. For our evaluation, we chose NoREC [25], TLP [26], DQE [31], and CODDTest [35] as baseline approaches. We selected these methods for several reasons. First, they have each successfully uncovered multiple bugs in widely used DBMSs, demonstrating their practical effectiveness. Second, they represent different directions in constructing testing oracles for logic bug detection. Specifically, NoREC and DQE represent the class of approaches that detect logic bugs by leveraging clause equivalence; TLP represents the set-relation–based oracle design. A similar work, Pinolo [13], also adopts this principle, but it does not support testing SQLite3, preventing a direct comparison. CODDTest represents the class of approaches that construct oracles by performing expression-equivalence transformations; a similar work, EET [15], adopts the same principle but cannot be run continuously—it terminates immediately after detecting a bug—making it unsuitable for long-running experiments. Another direction of oracle construction involves changing

configurations or optimizer hints to alter query execution paths for bug detection, as exemplified by Mozi [17], DQP [2], and TQS [33]. However, these tools do not support SQLite3 testing, and most of them mainly target join-related bugs, whereas EPSC is not limited to any specific syntactic feature. Finally, all five tools directly reuse SQLancer's test-case generation code, ensuring that the comparison of bug-detection capabilities focuses solely on the effectiveness of their testing oracles, while eliminating inconsistencies arising from differences in test-case generation implementations.

Results. The comparison results are shown in Table 2. Overall, although EPSC did not find the largest total number of unique bugs—its count is smaller than that of NoREC, TLP, and CODDTest, but higher than DQE—it ranks first in terms of the number of bugs that were exclusively detected by a single tool, tying with TLP and outperforming NoREC, CODDTest, and DQE. Figure 2 presents the overlap among the bugs identified by different testing methods. To avoid false negatives caused by their lack of support for prepared statements, we applied the baseline test oracles to the bug-inducing test cases of the 6 logic bugs only found by EPSC to examine whether they could detect the same inconsistencies. The results show that one of the six bugs can be detected by NoREC, TLP, and CODDTest, while the remaining five bugs cannot be detected by any of the baseline oracles. We then checked the fix commits corresponding to these five bugs and found that four of them are related to prepared statements. This result highlights the unique value of EPSC, demonstrating its effectiveness in discovering logic bugs that cannot be detected by existing methods.

Table 2. Effectiveness and efficiency of EPSC compared to state-of-the-art test oracles

Test oracles	# of unique bugs	# of oracle-exclusive bugs	Code coverage	# of unique query plans
NoREC	23	4	63.36%	172,812
TLP	24	6	63.86%	137,755
DQE	3	0	45.07%	475
CODDTest	21	2	62.33%	2,577,542
EPSC	8	6	69.06%	2,709,700

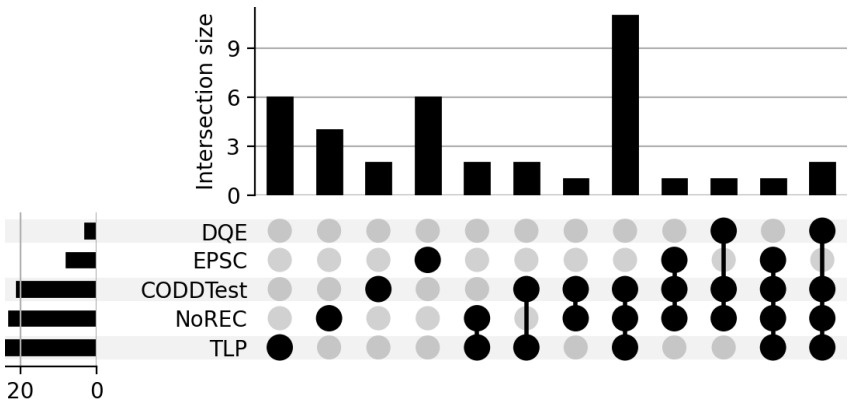


Fig. 2. Overlap among bugs detected by different methods.

Under identical statement generation settings, EPSC achieved higher code coverage across all tested DBMSs compared with other methods, improving coverage by up to 5.2% over the next best approach. This improvement primarily stems from its support for executing prepared statements, demonstrating that our approach effectively enables equivalent statements to exercise distinct execution paths within the DBMS. Furthermore, EPSC triggered the largest number of unique

query plans. This is largely attributed to its support for subqueries—similar to CODDTest—which increases query complexity and, consequently, plan diversity compared with NoREC and TLP. In addition, prepared statements were also able to trigger query plans that ordinary statements could not. During the 24-hour experiment, 2,319 unique query plans were observed exclusively from prepared statements.

In summary, EPSC is capable of exploring previously uncovered code regions and execution paths, and of revealing logic bugs that existing methods fail to detect.

4.4 Ablation Study

In our approach, we propose an error validation strategy to determine whether an logic bug occurs when inconsistent behavior is observed between equivalent ordinary statements and prepared statements. This strategy traverses the subexpressions within a query to identify erroneous parts that are skipped due to short-circuit optimization. The execution of redundant queries in this process is expected to degrade the overall performance of our method. However, in practice, we observed that this strategy had varying impacts across different DBMSs. In this experiment, we evaluate how this strategy affects both testing performance and the number of false positives.

Methodology. In this evaluation, we implemented EPSC$^-$, a variant of EPSC with error validation disabled. We conducted experiments on seven supported DBMSs, each running for 24 hours with 10 concurrent threads to minimize performance fluctuations caused by individual test runs. Three metrics are defined for this evaluation: the total number of tests executed within 24 hours (T), the number of alarms raised (A), and the average number of statements executed per test (SPT). Under the default setting of EPSC, step ① initializes the database state, and step ② is then executed up to 100,000 times, terminating early if a bug is detected. Each execution of step ② is regarded as a single test, as it performs a comparison between equivalent statements. On average, each test involves executing one ordinary statement, one prepared statement definition and execution, and, for systems like MySQL, additional statements for parameter definition. For EPSC, extra queries are executed for error validation. To compute SPT, we divide the total number of statements executed in steps ①, ②, and ③ during a test iteration by the number of executions in step ② (i.e., the number of tests). Thus, more executions of step ② should yield a smaller SPT.

Results. The results are shown in Table 3. Overall, EPSC produces up to 99.9% (i.e., on SQLite3) fewer bug reports than EPSC$^-$, because the error validation mechanism effectively reduces false positives. Nevertheless, some reports remain due to two factors: the latest DBMS releases do not yet include all bug fixes, and the matching conditions used to avoid duplicate reports are set conservatively to prevent missing newly introduced bugs.

Table 3. Effect of error validation step on the efficiency of EPSC and number of alarms.

Tool	MySQL			MariaDB			TiDB			PostgreSQL			CockroachDB			SQLite3			DuckDB		
	T	*A*	SPT	*T*	*A*	SPT	*T*	*A*	SPT	*T*	*A*	SPT	*T*	*A*	SPT	*T*	*A*	SPT	*T*	*A*	SPT
EPSC	499M	1,052	10.18	623M	25	10.42	138M	246	10.79	119M	1271	6.11	233M	15	5.59	4,719M	24	3.021	537M	94	5.39
EPSC$^-$	609k	220,381	148.74	622M	925	10.42	29M	83,142	12.16	16M	9,796	6.26	165M	8.709	5.72	4,543M	19,906	3.025	516M	19,821	5.73

We observed that EPSC performed more tests than EPSC$^-$ on all DBMSs. This is because EPSC$^-$ frequently raises bug reports, forcing it to repeatedly connect to the DBMS, create database instances, and run step ①. In numerous tests, iterations terminate early—either when DML statements trigger errors before step ② begins or before step ② reaches its maximum number of executions—resulting in an increased number of statements executed per test. This explains why EPSC$^-$ exhibits higher SPT than EPSC across all DBMSs. In client–server DBMSs, connecting to the server incurs significant performance overhead. In MariaDB, SQLite3, and DuckDB, the performance degradation of EPSC$^-$

compared with EPSC is relatively small. The generator of MariaDB is relatively simpler, and fewer abnormal test cases were produced. As a result, EPSC$^-$ generated fewer alarms, and the performance impact of the aforementioned process was smaller than the overhead introduced by error validation. In SQLite3 and DuckDB, connecting to a database instance is much faster, which makes EPSC$^-$ less affected by this issue. In this evaluation, although MySQL, MariaDB, and TiDB support largely similar SQL syntax, EPSC and EPSC$^-$ exhibit different results across these DBMSs. This difference arises because we directly adopt SQLancer's query generators, which are implemented separately for each DBMS and therefore produce SQL statements with different proportions of erroneous. Nevertheless, the overall trends of the results remain consistent.

To verify that all filtered reports by EPSC are indeed false positives, we deduplicated all error messages reported by EPSC$^-$ and conducted a manual analysis. This analysis revealed one genuine unexpected error. The error is skipped in the ordinary statement due to a false `WHERE` condition but is triggered in the prepared statement. During error validation, the expression is forcibly executed and triggers the same error, causing EPSC to classify it as a false positive. This error lies outside the detection scope of EPSC, as it is triggered both in the prepared and the ordinary statement. As with other metamorphic testing approaches, EPSC cannot detect bugs that manifest in both statements used to construct the oracle.

In summary, the error validation strategy effectively reduces false positives caused by inconsistencies in equivalent statement behavior due to short-circuit optimization in DBMSs. Moreover, by avoiding a large number of false positive reports, it also improves the overall testing performance.

5 Discussion

Testing scope. EPSC can test all statements that can be executed through prepared statements. Specifically, DML statements such as `INSERT`, `UPDATE`, and `DELETE`, as well as DQL statements like `SELECT`, are supported by almost all DBMSs to be executed in the form of prepared statements. In addition, based on the documentation of the tested DBMSs, we enabled testing for all implementations behind the language features that can be executed through prepared statements. For example, MySQL support executing certain DDL statements through prepared statements, and we have incorporated support for these cases in our implementation.[5]

Limitation. Although EPSC has uncovered multiple bugs, there are two types of bugs that it cannot detect, and other approaches based on metamorphic testing for logic bug detection face the same limitations. First, when a bug can be triggered by both an ordinary statement and a prepared statement, EPSC is unable to detect it. Second, EPSC cannot detect bugs in language features that exhibit nondeterministic behavior, such as the `RAND` function.

False positives. During implementation, EPSC generated three false positives. The first arose from short-circuit optimization skipping erroneous parts, causing inconsistent behavior. We eliminated these using our error-validation mechanism, successfully detecting logic bugs in DBMSs for erroneous statements and errors in correct statements. Take CockroachDB's results in Table 3 as an example, before error validation, 99.23% reports are false positives. After error validation, only 1 false positive remained. It involved an expression used in both `SELECT` and `GROUP BY`, binding a literal in this expression made the `GROUP BY` invalid. Error validation can not reproduce the same error as this is clause specific. We log the error message in `ClauseSpecificCheck` to avoid this false positive. The second occurred when using bound values exclusively in `GROUP BY` or `ORDER BY` clauses, leading to inconsistent results. To prevent this false positive, we avoid using bound values alone in `GROUP BY` and `ORDER BY` clauses across all DBMSs. The third appeared in PostgreSQL and CockroachDB when type hints were provided for literals in prepared statements but not for the

[5]https://dev.mysql.com/doc/refman/9.3/en/sql-prepared-statements.html

same literals in ordinary statements, resulting in different type inference and false positives. We added explicit type casts for literals in PostgreSQL and CockroachDB to address this.

6 Related Work

DBMS logic bug detection. Several approaches have been proposed to detect logic bugs in DBMSs, among which differential testing and metamorphic testing are the two primary categories. Differential testing [20] executes equivalent inputs across two or more comparable systems, with the expectation of producing equivalent outputs; any inconsistency may indicate the presence of a logic bug. By leveraging differential testing, RAGS [30] detects logic bugs by examining whether equivalent queries produce consistent results across different DBMSs or across versions of the same system. However, differential testing faces challenges in detecting logic bugs. On the one hand, SQL dialects differ across DBMSs, and differential testing has limited ability to handle system-specific features. On the other hand, logic bugs may arise within different versions of the same DBMS, making it difficult to uncover them by simply comparing results across versions.

Metamorphic testing [7] generates follow-up test cases from existing ones, expecting the outcomes of the original and follow-up cases to satisfy predefined metamorphic relations. NoREC [25] leverages the equivalence between expressions in the `SELECT` and `WHERE` clauses, assuming that an expression should evaluate to the same result in both contexts. DQE [31] extends NoREC by asserting that the same expression should evaluate consistently in the `WHERE` clause of `SELECT`, `UPDATE`, and `DELETE` statements. TLP [26] is based on the fact that an expression can evaluate to `TRUE`, `FALSE`, or `NULL`. It constructs three queries using `p`, `NOT p`, and `p IS NULL` as predicates, and asserts that their union should equal the complete set evaluated by the expression. Pinolo [13] similarly leverages set relationships to detect logic bugs by constructing looser or stricter predicates, such that the transformed query's result should either contain or be a subset of the original query's result. TQS [33] and Differential Query Plans (DQP) [2] detect logic bugs in join optimization by providing hints or modifying DBMS configurations to induce different query plans during query execution. Both EET [15] and CODDTest [35] transform expressions to construct equivalent queries. EET generates more complex but equivalent expressions by introducing subexpressions that always evaluate to `TRUE` or `FALSE`, while CODDTest creates simpler expressions by applying constant folding and propagation to replace parts of the original expression with equivalent constants.

Essentially, EPSC can also be regarded as a form of metamorphic testing. It constructs prepared statements that alter the parsing, compilation, optimization, and execution paths of their ordinary counterparts, while expecting both forms to produce identical results. Compared with differential testing approaches, EPSC is capable of detecting logic bugs that reside in DBMS-specific language features as well as those hidden across multiple versions of the same system. Compared with other metamorphic testing approaches, EPSC can be regarded as orthogonal to existing methods, as it effectively detects types of logic bugs that prior techniques fail to uncover. On the one hand, as shown in Section 4.3, during the 24-hour evaluation, six bugs detected by EPSC could not be identified by any other oracle. On the other hand, EPSC can detect bugs in DML statements. Compared with DQE, the only test oracle supporting DML statements—which can detect bugs only in the `WHERE` conditions of `UPDATE` and `DELETE` statements—EPSC can also detect bugs arising during the data manipulation process itself. Consequently, compared to prior work that constructs test oracles based on semantic equivalence or configuration equivalence, EPSC explores a new testing dimension—execution-mode equivalence—by contrasting prepared statements with their corresponding ordinary statements. By leveraging the implementation-level differences between these two execution modes, EPSC is able to effectively detect logic bugs that cannot be exposed by existing approaches.

DBMS test case generation. Numerous studies have focused on improving test case generation to enable fuzzing to cover more code regions and detect bugs more efficiently. These techniques can be considered complementary to EPSC, as they help improve the efficiency and effectiveness of bug triggering. Test case generation methods can be broadly categorized into generation-based and mutation-based approaches. Generation-based approaches produce syntactically and semantically valid test cases based on the grammar and features supported by the target DBMSs, with SQLSmith [28] and SQLancer [27] being representative examples. Query Plan Guidance (QPG) [1] was proposed to mitigate redundancy in randomly generated test cases and improve resource efficiency. It evaluates the sufficiency of testing for a given database state by analyzing the query plans triggered by executed queries. Once no additional queries can generate new query plans, the database state is mutated to further enhance testing coverage. QAGen [3] takes a query along with a set of constraints and constructs a database that satisfies the expected query results by integrating conventional query processing techniques with symbolic execution.

Mutation-based methods use existing test cases as seeds and derive new test cases through systematic transformations of the originals. SQLRight [18] proposes a coverage-guided approach to direct the mutation of test cases, enabling the generated queries to cover a larger portion of the code. Griffin [10] proposes a grammar-free approach for mutating test cases, summarizing the DBMS state into a metadata graph—a lightweight data structure that enhances the correctness of mutations in fuzzing and facilitates easy adaptation across different DBMSs. Sedar [9] proposes leveraging test cases from other DBMSs as seeds during fuzzing, in order to generate high-quality test inputs for the target DBMS.

Prepared statement optimization. Since the values of parameters in a prepared statement are unknown during compilation and optimization, optimizations applied to ordinary statements may not be directly applicable. Consequently, several approaches have been proposed to optimize prepared statements. Bizarro et al. [5] explores a query's parameter space to build parametric plans that often bypass the optimizer while still producing near-optimal query plans. Dutt et al. [8] point out that existing online approaches to parametric query optimization (PQO) struggle to balance sub-optimality, optimization overhead, and plan storage, and propose a plan re-costing method that addresses all three aspects effectively. Chaudhuri et al. [6] describes techniques for selecting query plans that better balance the trade-off between the average and variance of execution cost across different instances of a parameterized query. Vaidya et al. [34] decouples the generation and selection of query plans for prepared statements, proposing a novel algorithm to produce a set of plans that minimizes the optimizer's estimated query cost, and then employs a large language model to choose the optimal query plan for a specific prepared statement instance. These works demonstrate that the execution of prepared statements follows different code paths compared to ordinary statements, which provide the theoretical foundation for the effectiveness of our oracle.

7 Conclusion

This paper presents a general black-box approach for detecting bugs in DBMSs, termed EPSC. The key insight of EPSC is that a DML or DQL statement can be transformed into a prepared statement by replacing literal values with bound parameters. This transformation enables the construction of a test oracle that detects logic bugs as well as unexpected syntax or semantic errors by comparing the behavior and result sets of the two statement forms. EPSC has identified 52 unique, previously unknown bugs across seven mature and widely used DBMSs. Furthermore, the evaluation results demonstrate that EPSC effectively covers code paths and query plans that existing methods fail to explore, and detects logic bugs that they fail to identify. We believe that the simplicity and generality of EPSC can enhance the reliability of DBMSs.

Acknowledgments

We thank the reviewers for their valuable comments. This research was sponsored by the National Natural Science Foundation of China (No.U2441238, 625B2100), and the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No.JYB2025XDXM122).

References

[1] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2060–2071. doi:10.1109/ICSE48619.2023.00174

[2] Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proc. ACM Manag. Data* 2, 3, Article 188 (May 2024), 26 pages. doi:10.1145/3654991

[3] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. 2007. QAGen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China) *(SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 341–352. doi:10.1145/1247480.1247520

[4] Prithvi Bisht, A. Prasad Sistla, and V. N. Venkatakrishnan. 2010. TAPS: automatically preparing safe SQL queries. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '10)*. Association for Computing Machinery, New York, NY, USA, 645–647. doi:10.1145/1866307.1866384

[5] Pedro Bizarro, Nicolas Bruno, and David J. DeWitt. 2009. Progressive Parametric Query Optimization. *IEEE Trans. on Knowl. and Data Eng.* 21, 4 (April 2009), 582–594. doi:10.1109/TKDE.2008.160

[6] Surajit Chaudhuri, Hongrae Lee, and Vivek R. Narasayya. 2010. Variance aware optimization of parameterized queries. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 531–542. doi:10.1145/1807167.1807226

[7] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020).

[8] Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. 2017. Leveraging Re-costing for Online Optimization of Parameterized Queries with Guarantees. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1539–1554. doi:10.1145/3035918.3064040

[9] 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 the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 146, 12 pages. doi:10.1145/3597503.3639210

[10] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2023. Griffin: Grammar-Free DBMS Fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 49, 12 pages. doi:10.1145/3551349.3560431

[11] Kevin P Gaffney, Martin Prammer, Larry Brasfield, D Richard Hipp, Dan Kennedy, and Jignesh M Patel. 2022. SQLite: past, present, and future. *Proceedings of the VLDB Endowment* 15, 12 (2022).

[12] Ahmad Ghazal, Dawit Seid, Bhashyam Ramesh, Alain Crolotte, Manjula Koppuravuri, and Vinod G. 2009. Dynamic plan generation for parameterized queries. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 909–916. doi:10.1145/1559845.1559946

[13] 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 *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 345–358. https://www.usenix.org/conference/atc23/presentation/hao

[14] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[15] Zu-Ming Jiang and Zhendong Su. 2024. Detecting logic bugs in database engines via equivalent expression transformation. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 44, 15 pages.

[16] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.

[17] 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 the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 135, 12 pages. doi:10.1145/3597503.3639112

[18] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *Proceedings of the 31st USENIX Security Symposium (USENIX 2022)*. Boston, MA.
[19] MariaDB 2024. MariaDB. https://mariadb.org/. Accessed: March 19, 2026.
[20] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.
[21] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.
[22] MySQL 2024. MySQL. https://www.mysql.com/. Accessed: March 19, 2026.
[23] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*. 1981–1984.
[24] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: optimization of imperative programs in a relational database. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 432–444. doi:10.1145/3186728.3164140
[25] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1140–1152. doi:10.1145/3368089.3409710
[26] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (Nov. 2020), 30 pages. doi:10.1145/3428279
[27] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 667–682. https://www.usenix.org/conference/osdi20/presentation/rigger
[28] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. 2018. SQLsmith: a random SQL query generator. https://github.com/anse1/sqlsmith
[29] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 618–622.
[30] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. 618–622.
[31] 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 the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2072–2084. doi:10.1109/ICSE48619.2023.00175
[32] 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 the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. doi:10.1145/3318464.3386134
[33] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. *Proc. ACM Manag. Data* 1, 1, Article 55 (May 2023), 26 pages. doi:10.1145/3588909
[34] Kapil Vaidya, Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. 2021. Leveraging query logs and machine learning for parametric query optimization. *Proc. VLDB Endow.* 15, 3 (Nov. 2021), 401–413. doi:10.14778/3494124.3494126
[35] Chi Zhang and Manuel Rigger. 2025. Constant Optimization Driven Database System Testing. *Proc. ACM Manag. Data* 3, 1, Article 24 (Feb. 2025), 24 pages. doi:10.1145/3709674

Received October 2025; revised January 2026; accepted February 2026