Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation

Zu-Ming Jiang
Department of Computer Science
ETH Zurich

Zhendong Su
Department of Computer Science
ETH Zurich

Abstract

Database management systems (DBMSs) are crucial for storing and fetching data. To improve the reliability of such systems, approaches have been proposed to detect logic bugs that cause DBMSs to process data incorrectly. These approaches manipulate queries and check whether the query results produced by DBMSs follow the expectations. However, such *query-level manipulation* cannot handle complex query semantics and thus needs to limit the patterns of generated queries, degrading testing effectiveness.

In this paper, we tackle the problem using a fine-grained methodology—*expression-level manipulation*—which empowers the proposed approach to be applicable to arbitrary queries. To find logic bugs in DBMSs, we design a novel and general approach, *equivalent expression transformation (EET)*. Our core idea is that manipulating expressions of a query in a semantic-preserving manner also preserves the semantics of the entire query and is independent of query patterns. EET validates DBMSs by checking whether the transformed queries still produce the same results as the corresponding original queries. We realize our approach and evaluate it on 5 widely used and extensively tested DBMSs: MySQL, PostgreSQL, SQLite, ClickHouse, and TiDB. In total, EET found 66 unique bugs, 35 of which are logic bugs. We expect the generality and effectiveness of EET to inspire follow-up research and benefit the reliability of many DBMSs.

1 Introduction

Database management systems (DBMSs) are critical systems software and play important roles in modern data-driven applications and provide essential functionalities such as data storage and fetching [8, 12, 39]. Like other large-scale systems, DBMSs involve complicated code logic and various functionalities, and thus bugs are easily introduced during their development and maintenance [1, 13, 15]. One of the most critical kinds of bugs is *logic bugs*—the bugs silently cause DBMSs to produce incorrect query results [25–27]. To detect logic bugs, existing approaches generate SQL queries to test DBMSs and check whether the produced results follow the expectations [11, 25–27, 29, 35]. To do so, they either construct customized queries and validate the rows fetched by these queries [27], or transform the given queries and check whether the execution results of the transformed queries are consistent with the original ones [11, 25, 26, 29, 35].

However, all existing approaches have limited generality as they require the generated queries to follow specific patterns. Their generated queries cannot support SQL features that violate their designed query patterns, as shown in Figure 1. For example, PQS [27] requires that the results of the generated queries can be predicted by its manually implemented interpreter, and thus it is difficult for PQS to support advanced SQL features involving complicated calculations (*e.g.,* window functions). TLP [26] requires that the queries must contain predicates in `WHERE` or `HAVING` clauses for partitioning, while bug-triggering queries may not contain such clauses. In addition, TLP does not support advanced features like window functions and subqueries. DQE [29] limits its queries to only use common SQL features supported by `SELECT`, `UPDATE`, and `DELETE` statements, while any features (*e.g.*, `JOIN` operations and aggregate functions) supported by only one kind of statements are absent. Table 1 provides detailed information on whether existing approaches support specific SQL features. Except for our approach, none of the existing ones can encompass all the listed SQL features. Due to their limited support of general SQL queries, existing approaches miss many logic bugs (*e.g.*, the logic bug triggered by the query shown in Figure 2, which incorporate correlated subqueries and join operations).

The lack of generality in existing approaches is caused by the inherent limitations of their coarse-grained methodology, namely *query-level manipulation*. To construct customized queries or transform existing ones to other related queries, these approaches need to understand the semantics of the manipulated queries to guarantee the results produced by these queries follow their expectations. However, SQL queries are designed to be flexible [34], and can contain abundant and

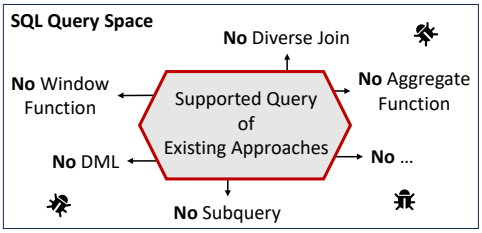

Figure 1: Limitations of existing approaches.

Table 1: Supported SQL features of existing approaches

Approach	Subquery	Join	Window	Group	DML
PQS	○	◐	○	●	○
NoREC	○	●	●	●	○
TLP	○	●	○	●	○
TQS	◐	●	○	●	○
Pinolo	◐	◐	○	○	○
DQE	○	○	○	○	●
EET	●	●	●	●	●

Window: window functions; Group: **`GROUP BY`** clauses; DML: data manipulation language (*e.g.*, **`UPDATE`** statements); ◐: TQS [35] and Pinolo [11] support simple but not correlated subqueries. PQS [27] and Pinolo support only parts of join operations.

complex semantics [13, 14, 16, 28, 37] supported by DBMSs. Existing approaches cannot be applied to complex queries because it is difficult to fully understand the complicated semantics contained by such queries (*e.g.*, the queries in Figure 2). Due to these challenges, existing approaches have to limit the patterns of their generated queries to constrain the query semantics. Therefore, these approaches cannot utilize general queries that fall outside their expected query patterns.

To address the inherent limitations of existing approaches, we propose to approach the logic-bug-detection problem using a new and fine-grained methodology—*expression-level manipulation*. Expressions are the essential units of SQL queries. They can be functions, operations, column variables, constant values, or subqueries, *etc*. By manipulating expressions, we can focus on the fine-grained semantics of queries, *i.e.*, expression semantics, and correspondingly manipulate queries without the need to understand their overall query-level semantics. For example, we can easily construct a new query for oracle checking by manipulating the expressions **`t2.c2`** and **`t2.c3`** of the original query in Figure 2, even though the query is complex. In this way, we do not need to limit query patterns to ones with simple semantics.

Based on this fine-grained methodology, we propose a novel and general approach, *equivalent expression transformation (EET)*, which applies to arbitrary queries and can effectively find logic bugs in DBMSs. Given an arbitrary query, EET traverses its abstract syntax tree (AST) to iterate

```
--- Statements for database generation
CREATE TABLE t0 (c0 TEXT);
CREATE TABLE t1 (c0 TEXT);
CREATE TABLE t2 (c0 INT4, c1 INT4, c2 TEXT,
                 c3 TEXT, c4 TEXT, c5 TEXT);
INSERT INTO t0 values ('');
INSERT INTO t1 values ('');
INSERT INTO t2 values (1, 2, 'a', 'a', 'a', 'a'),
                      (0, 1, '', '', 'a', 'L');

--- Original query, result set: 0 ✔
SELECT t2.c0 FROM t2
WHERE (t2.c1 >= t2.c0) <> (t2.c5 = (
   SELECT t2.c4 AS c_0
   FROM (t1 AS ref_0 INNER JOIN t0 AS ref_1
      ON (ref_0.c0 = ref_1.c0))
   WHERE t2.c3 = t2.c2
   ORDER BY c_0 DESC LIMIT 1));

--- Transformed query, result set: empty 🐞
SELECT t2.c0 FROM t2
WHERE (t2.c1 >= t2.c0) <> (t2.c5 = (
   SELECT t2.c4 AS c_0
   FROM (t1 AS ref_0 INNER JOIN t0 AS ref_1
      ON (ref_0.c0 = ref_1.c0))
   WHERE (CASE WHEN (((ref_0.c0 LIKE 'z~%')
            AND (NOT (ref_0.c0 LIKE 'z~%')))
            AND ((ref_0.c0 LIKE 'z~%') IS NOT NULL))
         THEN t2.c3 ELSE t2.c3 END) =
      (CASE WHEN (((ref_1.c0 NOT LIKE '_%%')
         AND (NOT (ref_1.c0 NOT LIKE '_%%')))
         AND ((ref_1.c0 NOT LIKE '_%%') IS NOT NULL))
      THEN t2.c4 ELSE t2.c2 END)
   ORDER BY c_0 DESC LIMIT 1));
```

Figure 2: Queries exposing an ancient logic bug (20 years old) in PostgreSQL.

over the expressions used in this query. For each expression, EET transforms it to another semantically equivalent one based on logical equivalences [5, 19] and SQL branch structures [34]. In the end, EET compares the execution results of the transformed query (*i.e.*, the query whose expressions have been transformed) and the original query, and any observed discrepancy indicates a logic bug. For example, EET transforms expressions **`t2.c2`** and **`t2.c3`** of the original query in Figure 2 into two semantically equivalent **`CASE WHEN`** expressions of the transformed query, and validate the tested DBMS by checking whether the results of these two queries are identical. The key intuition of what makes this approach effective is that the transformed expressions can lead DBMSs to exercise different code logic for the manipulated queries. Such different query executions cross-check each other as they are expected to produce the same results.

We implemented our approach as a practical DBMS testing tool and evaluate it on 5 widely-used and extensively-tested DBMSs, MySQL [20], SQLite [32], PostgreSQL [24], ClickHouse [3], and TiDB [36]. In total, EET found 66 unique bugs, including 16 in MySQL, 10 in SQLite, 9 in PostgreSQL, 21

in ClickHouse, and 10 in TiDB. Among these bugs, 65 are confirmed and 37 are fixed. 35 bugs are logic bugs, and many are long latent. These results demonstrate the effectiveness of EET in finding logic bugs in DBMSs.

Overall, we make the following contributions:

- We propose a fine-grained methodology, *expression-level manipulation*, which can operate on arbitrary queries without limiting query patterns.
- We propose a novel and general approach, *equivalent expression transformation (EET)*, which can effectively find logic bugs in DBMSs using transformation rules based on logical equivalences and SQL branch structures.
- We implement the approach as an automatic DBMS testing tool and evaluate it on 5 widely used DBMSs. In total, we found 66 bugs, 35 of which are logic bugs. To further facilitate research on DBMS testing, we open-source the tool at `https://github.com/JZuming/EET`.

2 Motivation

In this section, we illustrate queries that trigger an ancient logic bug, analyze the limitations of existing approaches, and present our solution based on a fine-grained methodology.

2.1 Illustrative Example

Figure 2 shows the queries that trigger a very ancient logic bug caused by incorrect hash-join mechanisms. The bug existed for 20 years in PostgreSQL until EET found it. The bug-triggering queries consist of three parts. The first part contains several statements (*e.g.*, **`CREATE`** and **`INSERT`**) for setting up a database for later querying. The second part is a randomly generated query, termed as *original query*, and the third part is the query transformed from the original query by our approach, term as *transformed query*. The expressions `t2.c3` and `t2.c2` in the original query are transformed into two semantically equivalent **`CASE WHEN`** expressions in the transformed query. The transformed query preserves the semantics of the original query, and thus these two queries should produce the same results. However, their results differed, indicating a logic bug had been triggered.

The bug-triggering queries have been minimized but are still very complex. Specifically, the transformed query contains a subquery in its **`WHERE`** clause. The subquery is a correlated subquery that references value (*i.e.*, the columns of table `t2`) from the outer query. The subquery uses **`INNER JOIN`** tables in its **`FROM`** clause and uses **`ORDER BY`** and **`LIMIT`** to constrain its returned value. PostgreSQL is expected to return a row `{0}` for the transformed query. However, it returns an empty set because the predicate inside the subquery triggers a logic bug. The detailed root cause is discussed in Section 5.3.

2.2 Limitations of Existing Approaches

Several approaches have been proposed to detect logic bugs in DBMSs [11, 25–27, 29, 35]. PQS [27] synthesizes a query in a way that the query is expected to fetch a specific row. If the row is not fetched, a logic bug is triggered. TLP [26] partitions a given query into three separated queries by decomposing the predicate in the **`WHERE`** or **`HAVING`** clause. The union of the results of these three queries should be consistent with the original one, otherwise, a logic bug is found. Pinolo [11] manipulates the query predicate in its **`WHERE`** clause to construct a new query whose results are the superset or subset of the results of the original query.

All these approaches are trapped in *query-level manipulation*, which is a coarse-grained methodology for logic-bug detection in DBMSs. To guarantee the correct manipulation, this methodology inherently requires the approaches to understand the semantics of the manipulated queries. For example, PQS needs to interpret its synthesized queries to predict their expected results. However, SQL is a flexible query language, providing various features (*e.g.*, subquery, join) to manipulate data [34]. Under specific demands, SQL queries (*e.g.*, analytical queries) can be very complex [16, 28, 37]. In these cases, query-level manipulation cannot work effectively because it cannot handle the complicated semantics of these queries.

For the example query in Figure 2, PQS cannot infer its expected fetched row because it cannot predict the results of the complex predicates involving correlated subquery with join tables. TLP cannot partition the predicates inside the subqueries, because predicate effects propagating from subqueries to the outer query are implicit and complicated. Partitioning predicates in subqueries cannot guarantee the consistency between the results of its unioned queries and its original query. Similarly, Pinolo cannot guarantee the superset or subset relationships between the manipulated queries because the logical effects inside the subqueries are difficult to predict.

To avoid such inapplicable cases from happening in their generated queries, existing approaches limit the query patterns to constrain the semantics of their generated query. As a result, many important SQL features cannot be supported by existing approaches, as shown in Table 1. For example, PQS and Pinolo support only parts of join operations, and none of the existing approaches supports correlated subqueries, because their semantics are complex. Therefore, these approaches miss many logic bugs that are not covered by their limited query patterns, such as the 20-year-old logic bug of PostgreSQL in Figure 2.

2.3 Our Solution

To propose a general approach that is applicable to arbitrary queries, we need to tackle the logic-bug-detection problem using a new methodology instead of query-level manipulation. In this paper, we propose a fine-grained methodology—

expression-level manipulation, which shifts our focus from the semantics of a whole query to the semantics of a single expression of the query, empowering the potential and flexibility of manipulation. We can manipulate queries by processing their fine-grained elements, expressions, without the necessity to analyze the semantics of the whole query, and thus do not need to limit the query patterns to simplify query semantics.

Based on this methodology, we propose *equivalent expression transformation (EET)*, which is applicable to arbitrary queries to find logic bugs in DBMSs. Given a SQL query, EET iterates all expressions of the query and transforms them into semantically equivalent expressions. EET validates the DBMSs by checking whether the query with transformed expressions produces the same results as the original query. In this paper, we use logical equivalences [5,19] and SQL branch structures [34] to perform semantic-preserving transformation. For example, in Figure 2, EET transform the expressions `t2.c3` and `t2.c2` in the original query to two **`CASE WHEN`** expressions in the transformed query. **`CASE WHEN`** expressions are SQL-style conditional branch structures. Depending on the results of the conditional expressions following **`WHEN`** keywords, the returned values of the **`CASE WHEN`** expressions are determined by the expression following either the **`THEN`** keywords (*i.e.*, **`TRUE`** branches) or **`ELSE`** keywords (*i.e.*, **`FALSE`** branches). Both the branch conditions of the two **`CASE WHEN`** are unsatisfiable and can only be evaluated to **`FALSE`**. Therefore, these two **`CASE WHEN`** are semantically equivalent to `t2.c3` and `t2.c2`, respectively, which are the expressions used in the original query. Therefore, the original query and the transformed query should produce the same results. However, the original query outputs 1 row `{0}`, while the transformed query outputs empty, exposing a logic bug.

EET is effective because the transformed expression can result in different execution logic of the tested DBMSs. For the example in Figure 2, the transformed expressions (*i.e.*, the two **`CASE WHEN`**) lead the PostgreSQL server to invoke its buggy hash-join mechanism, while the original one does not. Such execution differences make PostgreSQL produce different results for the two queries and indicate at least one of the queries triggers bugs.

3 Equivalent Expression Transformation

In this section, we present the formalization and overview of EET, the two kinds of expression transformations that we propose in this paper, and the properties of this approach.

3.1 Overview

We formalize the core idea of *equivalent expression transformation (EET)* as the following formula, where Q represents an arbitrary query, E represents expressions contained in Q,

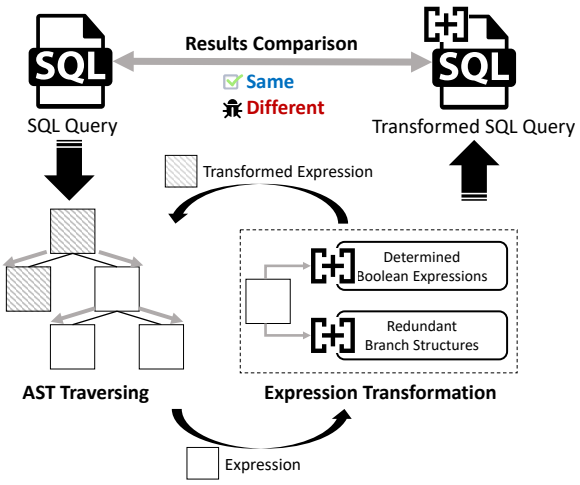

Figure 3: Approach overview of EET.

and $DB(Q)$ is the result the tested DBMS produces for Q:

$$E \equiv E' \Rightarrow DB(Q) \equiv DB(Q'), \text{where } Q' = Q[\![E'/E]\!] \quad (1)$$

The idea is simple: given an arbitrary query Q with expressions E, construct a query $Q' = Q[\![E'/E]\!]$ by replacing all occurrences of E in Q with semantically equivalent expressions E'. Q' and Q are semantically equivalent by construction, and a DBMS should produce identical results on them.

Figure 3 shows the overview of EET. EET traverses the AST presentation of the query to iterate over expressions and transforms these expressions into semantically equivalent ones. After all the expressions have been transformed, the EET constructs a semantically equivalent query and validates the tested DBMS by comparing the results of the transformed query and the original query. In this paper, we propose *determined boolean expressions* (Section 3.2.1) and *redundant branch structures* (Section 3.2.2) to instantiate the expression transformations that satisfy $E \equiv E'$ in Eq. 1.

3.2 Expression Transformation

SQL queries contain various expressions, whose types can be categorized into two classes: boolean expressions and non-boolean expressions. For boolean expressions, we can transform them leveraging the logical equivalences [5,19] in mathematical logic, which have been well studied and generally recognized. For non-boolean expressions, it is difficult to propose general transformation, because these SQL expressions can be numeric (*e.g.*, integer, floating point), string, or timestamp, *etc*. Their execution rules are different. To generally support these types, we leverage SQL branch structures [34], which operate on expressions of various types and provide flexibility for transformation.

To this end, we propose two kinds of expression transformation, *determined boolean expressions* and *redundant branch*

structures. Table 2 shows the details of each transformation, including its applied expressions and transformation rules. EET is extensible for additional expression transformations, and we expect that more kinds of effective transformation can be proposed in the future, as discussed in Section 3.3.

3.2.1 Determined Boolean Expressions

For each boolean expression, we can transform it using logical operations, such as **AND** and **OR**. We leverage 5 laws of logical equivalences [5, 19] for an arbitrary boolean expression p:

$$\top \vee p \equiv \top$$

$$\bot \wedge p \equiv \bot$$

$$\top \wedge p \equiv p$$

$$\bot \vee p \equiv p$$

$$p \vee q \equiv q \vee p, \;\; p \wedge q \equiv q \wedge p$$

We interpret them to corresponding SQL equations:

$$\texttt{TRUE OR } p \equiv \texttt{TRUE} \tag{2}$$

$$\texttt{FALSE AND } p \equiv \texttt{FALSE} \tag{3}$$

$$\texttt{TRUE AND } p \equiv p \tag{4}$$

$$\texttt{FALSE OR } p \equiv p \tag{5}$$

$$p \texttt{ OR } q \equiv q \texttt{ OR } p, \;\; p \texttt{ AND } q \equiv q \texttt{ AND } p \tag{6}$$

As the values of SQL boolean expressions can only be either **TRUE**, **FALSE**, or **NULL** [26], one of the expressions, p, **NOT** p, and p **IS NULL** must be **TRUE**, where p is an arbitrary boolean expression. Therefore, p **OR (NOT** p **)OR (** p **IS NULL)** must be **TRUE** according to Eq. 2. Similarly, one of the expressions, p, **NOT** p, and p **IS NOT NULL** must be **FALSE**, so p **AND (NOT** p **)AND (** p **IS NOT NULL)** must be **FALSE** according to Eq. 3. The equations are shown below:

$$p \texttt{ OR } (\texttt{NOT } p) \texttt{ OR } (p \texttt{ IS NULL}) \equiv \texttt{TRUE} \tag{7}$$

$$p \texttt{ AND } (\texttt{NOT } p) \texttt{ AND } (p \texttt{ IS NOT NULL}) \equiv \texttt{FALSE} \tag{8}$$

We use *true_expr* and *false_expr* to represent the expressions on the left-hand side of Eq. 7 and Eq. 8:

$$\mathit{true\_expr}(p) = p \texttt{ OR } (\texttt{NOT } p) \texttt{ OR } (p \texttt{ IS NULL}) \tag{9}$$

$$\mathit{false\_expr}(p) = p \texttt{ AND } (\texttt{NOT } p) \texttt{ AND } (p \texttt{ IS NOT NULL}) \tag{10}$$

Note that the operands of **OR**/**AND** can be randomly disordered according to Eq. 6 (*e.g.*, p and **NOT** p can switch their positions). Combining Eq. 4 with Eq. 7 and 9, Eq. 5 with Eq. 8 and 10, respectively, we get the following equations, where p and p' can be arbitrary boolean expressions:

$$\mathit{true\_expr}(p') \texttt{ AND } p \equiv p \tag{11}$$

$$\mathit{false\_expr}(p') \texttt{ OR } p \equiv p \tag{12}$$

Based on Eq. 11 and Eq. 12, we can transform an arbitrary boolean expression p by adding a structured expression containing a randomly generated boolean expression p'. We accordingly propose two transformation rules shown in rows No.1 and 2 in Table 2. Both of these transformation rules are guaranteed to preserve the semantics of original expressions. The queries shown in Figure 6 and Figure 7 (discussed in Section 5.3) are transformed by these rules.

3.2.2 Redundant Branch Structures

To transform non-boolean expressions, we propose to leverage **CASE WHEN** expressions, which are SQL-style conditional branch structures and support various SQL types. These expressions execute the following if-else logic:

$$C(p, \mathit{expr}_1, \mathit{expr}_2) = \texttt{CASE WHEN } p \texttt{ THEN } \mathit{expr}_1 \texttt{ ELSE } \mathit{expr}_2 \texttt{ END}$$

$$C(p, \mathit{expr}_1, \mathit{expr}_2) = \begin{cases} \mathit{expr}_1 & \text{if } p \text{ is TRUE} \\ \mathit{expr}_2 & \text{if } p \text{ is FALSE or NULL} \end{cases} \tag{13}$$

We can determine the execution logic of **CASE WHEN** expressions by fixing the predicate p to be **TRUE** or **FALSE**. Furthermore, we can use Eq. 9 and Eq. 10 to replace the **TRUE** and **FALSE** values, making the transformed expression more complex. In the end, we get the following equations:

$$C(\mathit{true\_expr}(p), \mathit{expr}, \mathit{expr}') \equiv \mathit{expr} \tag{14}$$

$$C(\mathit{false\_expr}(p), \mathit{expr}', \mathit{expr}) \equiv \mathit{expr} \tag{15}$$

Based on Eq. 14 and Eq. 15, we can transform an expression to a designed **CASE WHEN** expression, which involves randomly generated expressions p and expr' but still preserves the semantics of the original expression *expr*. We accordingly propose 2 transformation rules shown in rows No.3 and 4 of Table 2. Note that the type of expr' should be the same as the type of *expr*, otherwise ambiguous behaviors may be triggered in some DBMSs [20, 24, 32]. The query shown in Figure 2 is transformed by these rules.

Besides fixing the predicate p to be **TRUE** or **FALSE** to determine the return value of a **CASE WHEN** expression, we can also manipulate the expression in **TRUE** or **FALSE** branch. If the expressions in the **TRUE** and **FALSE** branch are semantically equivalent to each other, the **CASE WHEN** expression is determined to be the expression in the **TRUE**/**FALSE** branch, no matter how the predicate p is evaluated. Formally, we get the following equation by making $\mathit{expr}_1 \equiv \mathit{expr}_2$ in Eq. 13:

$$\mathit{expr}_1 \equiv \mathit{expr}_2 \Rightarrow C(p, \mathit{expr}_1, \mathit{expr}_2) \equiv \mathit{expr}_1 \equiv \mathit{expr}_2 \tag{16}$$

Based on Eq. 16, we propose 2 transformation rules shown in rows No.5 and 6 of Table 2. In these rules, we deeply copy

Table 2: Transformation rules of EET

No.	Expression Transformation	Applied Expression	Transformation Rule
1	Determined Boolean Expressions	*bool_expr*: boolean	*bool_expr* → *false_expr*[3] **OR** *bool_expr*
2	Determined Boolean Expressions	*bool_expr*: boolean	*bool_expr* → *true_expr*[4] **AND** *bool_expr*
3	Redundant Branch Structures	*expr*: **CASE-WHEN** applicable	*expr* → **CASE WHEN** *false_expr* **THEN** *rand_expr(type(expr))*[1, 2] **ELSE** *expr* **END**
4	Redundant Branch Structures	*expr*: **CASE-WHEN** applicable	*expr* → **CASE WHEN** *true_expr* **THEN** *expr* **ELSE** *rand_expr(type(expr))* **END**
5	Redundant Branch Structures	*expr*: **CASE-WHEN** applicable	*expr* → **CASE WHEN** *rand_expr(boolean)* **THEN** *copy_expr(expr)*[5] **ELSE** *expr* **END**
6	Redundant Branch Structures	*expr*: **CASE-WHEN** applicable	*expr* → **CASE WHEN** *rand_expr(boolean)* **THEN** *expr* **ELSE** *copy_expr(expr)* **END**
7	Origin	*expr*: non boolean **CASE-WHEN** inapplicable	*expr* → *expr*

[1] *type(e)*: the type of the return value of expression *e*.
[2] *rand_expr(t)*: randomly generated expression that returns a value with type *t*.
[3] *false_expr* → *p* **AND (NOT** *p* **)AND (** *p* **IS NOT NULL)** | *p* → *rand_expr(boolean)*
[4] *true_expr* → *p* **OR (NOT** *p* **)OR (** *p* **IS NULL)** | *p* → *rand_expr(boolean)*
[5] *copy_expr(e)*: an expression deeply copied from expression *e*

the original expression *expr* (*i.e.*, the expression itself and its subexpressions) to another expression, namely *copy_expr*. The expressions *expr* and *copy_expr* are semantically equivalent because they are the same. We distribute these expressions in **TRUE** and **FALSE** branches of a **CASE WHEN** expression, and randomly generate a predicate. Eq. 16 guarantees that such a **CASE WHEN** expression is semantically equivalent to the original expression. The query shown in Figure 8 (discussed in Section 5.3) is transformed by these rules.

3.2.3 Choosing Transformation Rules

CASE WHEN expressions can be applied for the majority types of SQL expressions, including numeric types, string types, timestamp types, *e.t.c.* EET randomly chooses one of the rules No.3 to 6 to transform the expression belonging to these types. Boolean types also support **CASE WHEN**, so EET randomly applies one of the rules No.1 to 6 for each boolean expression. However, some types of expressions are **CASE-WHEN** inapplicable, and thus none of the rules No.1 to 6 are available. The table expressions `t0` and `t1` in Figure 2 are the examples. When we replace them with **CASE WHEN** expression, the query will trigger syntactic errors. To address this problem, EET conservatively transforms these expressions to themselves, as shown in rule No.7 of Table 2.

3.3 Properties

Soundness. EET follows the formally-proved equations in Section 3.2, and thus is guaranteed to preserve the semantics of the original queries. If the execution results of the original queries are determined (*i.e.*, excluding SQL features involving randomness), the transformed queries must produce the same execution results as the original ones, otherwise logic bugs are triggered. Therefore, EET is sound and produces no false positives in logic-bug detection.

Generality. EET can be generally applied to various SQL queries because it works at the expression level instead of the query level. Existing approaches, which work at the query level, inherently require the generated queries to follow specified query patterns. Otherwise, these approaches cannot infer the oracle results to validate the execution of their generated queries. Such limitations make existing approaches not general because they cannot be applied to arbitrary queries (*e.g.*, the queries violated their patterns). In contrast, EET can be applied to validate arbitrary queries because it works at the expression level. Given an arbitrary query that is not limited to any query patterns, EET can generally transform it by transforming its expressions, and use the results of the transformed query to validate the results of the given query. In this sense, EET is general in validating arbitrary SQL queries.

While the high-level idea of EET works generally, its im-

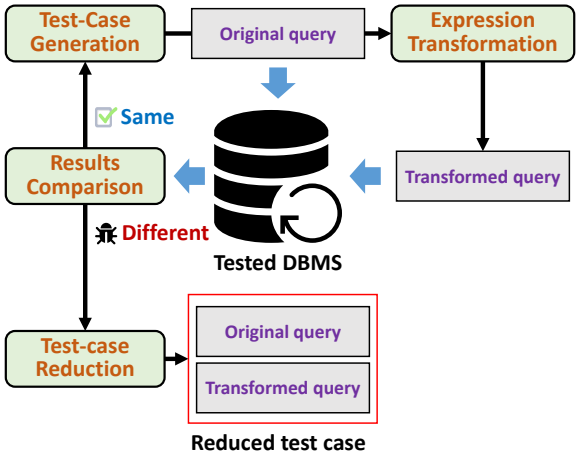

Figure 4: Implementation architecture of EET.

plementation for different SQL dialects can vary, depending on their branching structure syntax. For example, the function `DECODE` in Oracle [21] can return different values according to the comparison of its operands. This function can be implemented in the redundant branch structures of EET to transform queries written in Oracle-style SQL. In this paper, for redundant branch structures, we use only `CASE WHEN` expressions, which are supported by all the tested DBMSs.

Extensibility. In this paper, we propose two kinds of expression transformations for finding logic bugs to demonstrate the effectiveness of EET. Besides these two, we expect that more transformations are possible to enhance the approach. For example, a new transformation can be proposed to process table expressions (*e.g.*, `t0` and `t1` in Figure 2) by joining the original table with other tables while keeping the join results the same as the original one. EET can be easily extended to support new transformations as we only need to specify the transformation rules and the applied expression types. In our implementation, we used less than 200 lines of code for the proposed two kinds of expression transformations.

Black-Box Testing. EET is a pure black-box technique, which does not rely on the internal implementation of the tested DBMSs. Such a property makes our approach portable and can be easily deployed for testing various DBMSs, even the ones whose source code is unavailable.

4 Implementation

We implemented EET into a fully automatic tool on top of SQLsmith [33], which we mainly use for generating databases and complex queries. The overall codebase of the tool is 14k lines of C/C++ code, 2k of which is used to implement our approach. Figure 4 shows its architecture. The following describes the important implementation details.

Test-Case Generation. EET generates databases and queries randomly. To generate a query, the tool incrementally builds

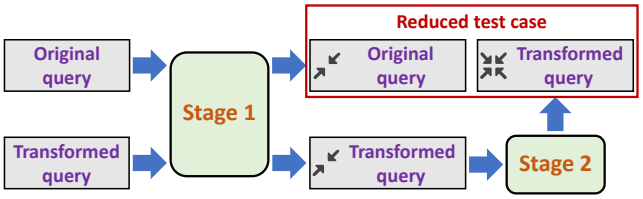

Figure 5: Two-stage reduction of EET.

an AST tree according to the grammar of SQL [27, 34, 38]. When constructing a node of the AST, EET updates and records the available variables (*e.g.*, relations, columns), and fills the node with a randomly generated expression referencing the available variables. After the AST tree is completed, a new query is generated and can be fed to the tested DBMS.

Expression Transformation. EET leverages the AST representation of the query to iterate each expression. For each expression, EET checks its type and randomly chooses one of the suitable transformation rules, as discussed in Section 3.2.3. During transformation, EET may need to generate additional expressions (*e.g.*, an additional boolean expression is required in rules No.1 to 6 in Table 2). In this case, EET reuses the information (*e.g.*, available variables in corresponding AST nodes) used in test-case generation to randomly generate additional expressions with specific types.

Result Comparison. EET compares the execution results of the transformed query and the original query, including their query output and database changes caused by these queries. Any discrepancy indicates that logic bugs are triggered.

Test-case Reduction. To trigger a logic bug, EET may perform transformations on a large query (*e.g.*, hundreds of lines of SQL), and all its expressions are transformed. However, the same bug might be triggered using parts of the large query and couples of transformed expressions. To ease the burden of developers, we need to reduce the bug-triggering queries before reporting the bugs. We customize *two-stage reduction* for EET to reduce the bug-triggering queries automatically. Figure 5 shows the workflow of this reduction procedure.

In the first stage, EET reduces both the original query and the transformed query. Each time the parts (*e.g.*, an expression) of an original query are reduced, the corresponding parts (*e.g.*, the transformed expression) in the transformed query are also reduced to make these two queries consistent. For example, in Figure 2, when the expression `t2.c3` of the original query is reduced to a constant value `NULL`, the corresponding `CASE WHEN` expression in the transformed query should also be replaced by `NULL`. EET checks whether these two queries still produce different results. If so, the bug still exists and the reduction is effective. Otherwise, EET recovers the reduced parts and tries to reduce other parts of these two queries. When no part can be reduced, EET enters the second stage.

In the second stage, EET tries to incrementally disable the

transformation for each expression in the transformed queries. After one transformation for an expression is disabled, if the execution results of the transformed queries are still different from the original, EET keeps this expression not transformed. Otherwise, EET recovers the expression to the transformed version. When no transformation can be disabled in the transformed query, this stage ends.

5 Evaluation

Our evaluation aims to answer the following questions to demonstrate the effectiveness of EET:

Q1 Can EET find real logic bugs in widely used and extensively tested DBMSs? (Section 5.2)

Q2 How diverse are the logic bugs found by EET? (Section 5.3)

Q3 Can EET find logic bugs that are missed by existing approaches? (Section 5.4)

5.1 Experimental Setup

We focused on testing open-source DBMSs for transparent and convenient bug reporting. We chose MySQL [20], PostgreSQL [24], SQLite [32], ClickHouse [3], and TiDB [36] because they are popular and extensively tested. At the time of paper writing, MySQL, PostgreSQL, and SQLite rank 1st, 2nd, and 6th, respectively, among the open-source DBMSs according to their popularity in DB-Engines Ranking [6]. ClickHouse and TiDB are relatively new DBMSs but very popular on GitHub [10]. They have gained over 31K and 35K stars, respectively, demonstrating their popularity. All of these DBMSs have been extensively tested by existing approaches for finding logic bugs [4, 11, 25–27, 29, 35] and crash bugs [13, 33, 38]. Finding new bugs in these DBMSs is very challenging and can demonstrate the effectiveness of EET.

We use EET to test the latest version of each DBMS. When the code of a DBMS is updated, we start a new test for the updated version. Specifically, we started to test MySQL from version 8.0.34, PostgreSQL from commit `3f1aaaa`, SQLite from commit `20e09ba`, ClickHouse from commit `30464b9`, and TiDB from commit `f5ca27e`. All the DBMS code is cloned from their official GitHub repositories. We intermittently deployed EET to test these DBMSs. We stopped and restarted the testing when we implemented new features in EET. The overall testing duration is three months. We evaluate EET on Ubuntu 20.04 with a 64-core AMD Epyc 7742 CPU at 2.25G Hz and 256GB RAM.

5.2 Bug Detection

As shown in Table 3, we reported 66 unique DBMS bugs found by EET, including 16 in MySQL, 9 in PostgreSQL, 10 in SQLite, 21 in ClickHouse, and 10 in TiDB. 65 of these bugs have been confirmed, and 37 have been fixed. None of these bugs are marked as duplicates by developers.

Table 3: Status of the bugs found by EET

DBMS	Reported	Confirmed	Fixed
MySQL	16	16	2
PostgreSQL	9	9	8
SQLite	10	10	10
ClickHouse	21	20	15
TiDB	10	10	2
Total	66	65	37

Table 4: Bug classification

DBMS	Logic	Crash	Error
MySQL	10	6	0
PostgreSQL	3	3	3
SQLite	9	0	1
ClickHouse	11	3	7
TiDB	2	7	1
Total	35	19	12

Bug Classification. We classify the bugs EET found into the three following types:

Logic bugs. The tested DBMSs incorrectly execute the SQL queries and produce wrong results (*e.g.*, select or update incorrect rows). These bugs were exposed because they incurred discrepancies between the results of the original queries and the transformed queries generated by EET.

Crash bugs. These bugs cause the tested DBMSs to crash or panic when specific queries are processed. Their root causes may be (1) memory corruption like null-pointer dereference; (2) assertion failures and (3) unexpected memory exhaustion.

Abnormal errors. The tested DBMSs report unexpected errors (*e.g.*, "database disk image is malformed" in SQLite) when processing syntactically and semantically valid queries.

As shown in Table 4, 35 bugs (52% of the bugs EET found) are logic bugs, which are the most interesting and hard-to-find bugs. The three logic bugs in PostgreSQL are exciting because PostgreSQL is a well-known hard nut for DBMS testing [13, 25, 27], where SQLancer [30], the DBMS testing tool integrating three logic bug detection techniques [25–27], found only one logic bug [31]. EET also found 19 crash bugs and 11 abnormal errors. These results demonstrate that EET is effective in finding bugs in DBMSs, especially logic bugs.

Bug Importance. We collect the severity information of the bugs we reported to MySQL and TiDB. PostgreSQL, SQLite, and ClickHouse do not provide the severity of each reported bug. All the crash bugs we reported to MySQL were identified as *confidential*, among which 2 have been assigned CVEs.

Among the 10 logic bugs in MySQL, 7 were recognized as *serious* bugs, and 3 were *non-critical*. Among the 10 bugs reported to TiDB, developers marked 6 as *major*, 1 as *minor*, and 3 as *moderate*.

The developers appreciated our effort in finding real bugs in their DBMSs. Particularly, PostgreSQL developers recognized our contribution to the reliability of PostgreSQL and sent us their commemorative coin [23]. ClickHouse and PostgreSQL developers provided their testimonials:

> ***ClickHouse:*** This tool has proven its value, and we want to integrate it into our CI and use it continuously. Thanks to @xxx for running it and reporting the findings!

> ***PostgreSQL:*** Thanks for your efforts! I thought about the generation of self-join tests for about a year, and it would be interesting to read about your approach. Could you send me a copy of the paper after release? Or the name of the conference to participate and see it offline.

Throughput. We count the number of tests (each one consists of one original query and one transformed query) performed by EET per second during testing the 5 DBMSs. On average, a single EET instance performs 3.39 tests per second (293k tests per day), which is lower than existing approaches. It is reasonable because EET supports complex queries, and DBMSs spend much more time executing complex queries than simple queries [13], making most CPU time spent on query execution (94.18% in our statistic results). We believe this throughput is practical considering (1) DBMS testing typically persists for several months [25–27, 29], and thus a sufficient number of tests can be executed and (2) setting up multiple testing instances can significantly improve the test efficiency of EET.

Table 5: SQL features of the 35 queries triggering logic bugs

ID	DBMS	Subquery	Join	Window	Group	DML
1	MySQL	●	●	○	○	○
2	MySQL	○	○	○	○	○
3	MySQL	○	○	○	○	○
4	MySQL	●	●	○	○	○
5	MySQL	●	○	●	○	○
6	MySQL	○	○	○	○	○
7	MySQL	○	○	○	○	○
8	MySQL	●	○	●	○	○
9	MySQL	○	○	○	○	○
10	MySQL	○	○	○	○	○
11	PostgreSQL	○	●	○	●	○
12	PostgreSQL	●	●	○	○	○
13	PostgreSQL	●	●	○	○	○
14	SQLite	●	●	○	○	○
15	SQLite	●	●	○	○	○
16	SQLite	●	○	○	○	●
17	SQLite	●	○	○	○	●
18	SQLite	●	●	●	○	○
19	SQLite	●	●	○	○	○
20	SQLite	○	●	○	○	○
21	SQLite	○	●	○	●	○
22	SQLite	○	●	○	○	○
23	ClickHouse	○	○	○	○	○
24	ClickHouse	●	●	○	○	○
25	ClickHouse	○	○	○	○	○
26	ClickHouse	●	●	○	○	○
27	ClickHouse	●	●	○	●	○
28	ClickHouse	●	○	●	○	○
29	ClickHouse	○	○	○	○	○
30	ClickHouse	●	○	○	○	○
31	ClickHouse	○	○	○	○	○
32	ClickHouse	○	●	○	○	○
33	ClickHouse	○	●	○	○	○
34	TiDB	●	○	○	○	○
35	TiDB	○	●	○	○	○

5.3 Bug Diversity

For the 35 logic bugs, we investigate their diversity from three aspects: (1) the diversity of bug-triggering queries involving multiple SQL features, (2) the diversity of the root causes of why DBMS produces incorrect results, and (3) the diversity of bug manifestation during testing DBMSs.

SQL Features. Table 5 lists the SQL features used in the 35 queries triggering logic bugs. The columns *Subquery*, *Join*, *Window*, and *Group* show whether the queries contain subqueries, join operations, window functions, and `GROUP BY` clauses, respectively. The column *DML* shows whether the bug-triggering queries are DML statements (*e.g.*, `UPDATE` and `DELETE`) instead of DQL (*e.g.*, `SELECT`).

Among the 35 bug-triggering queries, 18 contain subqueries (8 of them involve correlated subqueries), 18 use join operations (*e.g.*, inner join, outer join, and cross join), 4 invoke window functions (*e.g.*, `DENSE_RANK`, `FIRST_VALUE`), 3 involve `GROUP BY` clauses, and 2 are DML statements. We investigated the 10 bugs not involving these 5 features and found that all of them used SQL functions (*e.g.*, `ACOS`, `HEX`, and `UNIX_TIMESTAMP`) to perform complicated value calculations, string manipulation, and timestamp controlling. These results indicate that EET can find logic bugs triggered by various SQL queries.

The combined results in Table 1 and Table 5 demonstrate that EET can find many logic bugs missed by existing approaches because EET can support more various SQL features. For example, PQS [27], TLP [26], and NoREC [25] cannot find the 18 logic bugs related to subqueries, which are not supported by these approaches. Lacking support for join operations, DQE [29] cannot find the 18 join-related logic

```
--- Statements for database generation
CREATE TABLE t0 (c0 INT , c1 INT, c2 INT);
INSERT INTO t0 VALUES(2,1,-20);
INSERT INTO t0 VALUES(2,2,NULL);
INSERT INTO t0 VALUES(2,3,0);
INSERT INTO t0 VALUES(8,4,95);
--- Original query, delete 4 rows ✔
DELETE FROM t0 WHERE TRUE;
--- Transformed query, delete 3 rows 🐞
DELETE FROM t0 WHERE
((((t0.c0 <= t0.c2) AND
   (t0.c0 <> (SELECT c0 FROM t0 ORDER BY c0 LIMIT 1
   OFFSET 2))) IS NULL) OR
((t0.c0 <= t0.c2) AND
   (t0.c0 <> (SELECT c0 FROM t0 ORDER BY c0 LIMIT 1
   OFFSET 2))) OR
NOT ((t0.c0 <= t0.c2) AND
   (t0.c0 <> (SELECT c0 FROM t0 ORDER BY c0 LIMIT 1
   OFFSET 2))))AND TRUE;
```

Figure 6: Queries triggering a logic bug in the one-pass optimization of SQLite.

bugs. Notably, none of the existing approaches could find the 8 logic bugs related to correlated subqueries, as they cannot support semantically complex features. EET, benefiting from expression-level manipulation, is not limited to specific query patterns and can generally support all the listed features. Therefore, EET can find many bugs beyond the capabilities of existing approaches.

Root Cause Analysis. We investigated the 19 fixed logic bugs, whose patches and developer feedback are visible. They consist of 9 bugs in SQLite, 3 in PostgreSQL, and 7 in ClickHouse. We found that 12 bugs are caused by incorrect optimization. It is expected because EET supports logic bug detection for complex queries, which has huge potential to be optimized and thus can cover many optimization mechanisms in the tested DBMSs. 11 bugs are related to **`JOIN`** operations. Indeed, existing approaches could not systematically test the DBMS components related to **`JOIN`** operations until TQS was proposed [35], and thus many bugs remain unexposed. These results indicate that EET can also be used to effectively test the **`JOIN`** mechanism implemented in the tested DBMSs (*e.g.*, the hash-join bug shown in Figure 2). Different from TQS, EET can also detect bugs in other DBMS components, such as a bug in the JIT components used for expression compilation in ClickHouse. The following shows three interesting bug examples caused by different root causes.

Example 1: Optimization bug in SQLite. Figure 6 shows the queries triggering a logic bug in the one-pass optimization of SQLite. The original query is a simple **`DELETE`** statement with a predicate **`TRUE`**, which removes 4 rows in the table **`t0`**. EET transforms this query by applying rule No.2 in Table 2 to the predicate **`TRUE`**, which is semantic preserving. The transformed query should also remove 4 rows in **`t0`**, but only 3 rows are removed, indicating a logic bug triggered. The root

```
--- Statements for database generation
CREATE TABLE t0 (c0 UInt32, c1 UInt32,
        PRIMARY KEY (c0)) ENGINE = MergeTree;
INSERT INTO t0 VALUES (2, 2);
--- Original query, result set: {FALSE} ✔
SELECT FALSE FROM t0;
--- Transformed query, result set: {TRUE} 🐞
SELECT (acos(c0) <> atan(c1)) AND
         (NOT (acos(c0) <> atan(c1))) AND
         ((acos(c0) <> atan(c1)) IS NOT NULL)
       OR FALSE from t0;
```

Figure 7: Queries triggering a logic bug in the JIT compilation of ClickHouse.

cause is that the transformed query triggered the one-pass optimization in SQLite, which passes the target table only one time. For each row, SQLite evaluates whether it satisfies the predicate. If so, SQLite deletes the row. Because the subquery in the **`WHERE`** clause is behind a short-circuit operator (*i.e.*, **`AND`** operation), SQLite evaluates it after one or more rows have already been deleted, and SQLite thus produces a wrong result for the subquery, which at the end makes a row in table **`t0`** not deleted. SQLite developers fix this bug by disabling the one-pass optimization when the processed query contains a subquery in its **`WHERE`** clause.

Example 2: JOIN bug in PostgreSQL. Figure 2 shows the query triggering an ancient logic bug in the hash-join implementation of PostgreSQL. Specifically, PostgreSQL built a hash table for the **`INNER JOIN`** tables (*i.e.*, **`t1`** and **`t0`**) in the **`FROM`** clause. Their hash table was affected by a PostgreSQL data structure, *Param*, a parameter used for passing values into and out of subqueries or from nested loop joins to their inner scans. PostgreSQL must rebuild the hash table when specific Param values are updated. However, in some cases, the inner hash-key expressions for the hash table reference some Params whose changes are unexpectedly missed by PostgreSQL. The transformed query in Figure 2 updated such Params while PostgreSQL did not perceive the changes of these Params, as a result of which PostgreSQL incorrectly reused the outdated hash table and produced wrong results. PostgreSQL developers fixed this bug by invoking specific functions to take the missed Params into account.

Example 3: JIT bug in ClickHouse. Figure 7 shows the query triggering a logic bug in the JIT compilation of ClickHouse. The original query is a simple **`SELECT`** statement. EET transforms the **`FALSE`** expression in the **`SELECT`** clause, applying rule No.1 in Table 2. The transformed query trigger the JIT compilation of ClickHouse because the query repeatedly uses the expression **`acos(c0)<> atan(c1)`**. To speed up the query execution, ClickHouse compiles this expression into machine code that can be executed by CPUs directly. However, the JIT compiler incorrectly compiled the non-equal operation (*i.e.*, **`<>`**), and thus the machine code produced wrong

```
--- Statements for database generation
CREATE TABLE t0 (c0 INT, c1 INT);
CREATE TABLE t1 (c0 INT, c1 INT, c2 REAL,
                 c3 REAL, c4 INT);
INSERT INTO t0 VALUES(14, 24000);
INSERT INTO t1 VALUES(85, 95000, 97.87, 0.0, 0);
--- Original query, result set: {1}
SELECT DISTINCT 1 AS c1
FROM ((t1 AS ref_0 RIGHT OUTER JOIN t0 AS ref_1
         ON ref_0.c4 = ref_1.c0)
   LEFT OUTER JOIN
      (t1 AS ref_2 LEFT OUTER JOIN t0 AS ref_3
         ON ref_2.c1 = ref_3.c0)
   ON (((SELECT c1 FROM t0 ORDER BY c1 LIMIT 1) IN (
     SELECT ref_4.c0 AS c0 FROM t1 AS ref_4)) IS TRUE))
WHERE ref_2.c3 <= ref_2.c2;
--- Transformed query, result set: empty
SELECT DISTINCT 1 AS c1
FROM ((t1 AS ref_0 RIGHT OUTER JOIN t0 AS ref_1
         ON ref_0.c4 = ref_1.c0)
   LEFT OUTER JOIN
      (t1 AS ref_2 LEFT OUTER JOIN t0 AS ref_3
         ON ref_2.c1 = ref_3.c0)
   ON (((SELECT c1 FROM t0 ORDER BY c1 LIMIT 1) IN (
     SELECT ref_4.c0 AS c0 FROM t1 AS ref_4)) IS TRUE))
WHERE CASE WHEN TRUE THEN ref_2.c3 <= ref_2.c2
           ELSE ref_2.c3 <= ref_2.c2 END;
```

Figure 8: SQLite logic bug triggered by the original query.

results when comparing the **NaN** value returned from **acos(c0)**. Therefore, the transformed query produces an unreasonable result. The developers fix this bug by repairing the function responsible for compiling the non-equal operation.

Bug Manifestations. While analyzing the 19 fixed logic bugs, we found another interesting phenomenon: a logic bug can be triggered by either the transformed query, the original query, or both of them. Specifically, 10 of the 19 bugs are triggered by the transformed queries (*e.g.*, Example 1-3), while 8 bugs are triggered by the original queries (*e.g.*, Example 4). Interestingly, EET found a logic bug triggered by both the transformed query and the original query (*i.e.*, Example 5), whose results are different. These results demonstrate that EET can catch a logic bug if any discrepancy is incurred between their result, independently of which query is the culprit.

Example 4: Bug triggered in the original queries. Figure 8 shows a case where the original query triggers a bug in SQLite. The original query contains the **DISTINCT** keyword, and its **FROM** clause consists of multiple join tables with many join conditions specified in the corresponding **ON** clauses, while the predicate in the **WHERE** clause is a simple comparison expression. In this case, SQLite applies the omit-outer-join optimization, which can reduce the useless join tables (*e.g.*, **ref_3**) that are not referenced outside their **JOIN** expressions. However, this optimization did not work well when SQLite also flattened the subqueries in the **JOIN** expressions (*e.g.*, the two subqueries in the last **ON** clause). As a result, SQLite incorrectly reduced the tables consisting of the flattened subqueries and thus produced wrong results for the original query. EET transforms the predicate in the **WHERE** clause of the original query to a **CASE WHEN** expression. Such transformation makes the predicate complex and blocks the buggy omit-outer-join optimization, and thus SQLite produced correct results for the transformed query. SQLite developers fix this bug by adding restrictions for applying omit-outer-join optimization.

```
--- Statements for database generation
CREATE TABLE t0 (c0 TEXT);
CREATE TABLE t1 (c0 INT4, c1 INT4, c2 TEXT,
                 c3 INT4, c4 FLOAT8, c5 INT4);
CREATE TABLE t2 (c0 TEXT, c1 TIMESTAMP);
CREATE VIEW t3 AS
 SELECT '1' AS c_0
   FROM ((SELECT ref_0.c0 AS c_0 FROM t0 ref_0
          GROUP BY ref_0.c0) subq_0
     FULL JOIN t2 ref_1 ON (subq_0.c_0 = ref_1.c0))
  WHERE ref_1.c1 > ref_1.c1;
CREATE VIEW t4 AS
 SELECT ref_1.c5 AS c_2, ref_1.c4 AS c_3,
          ref_1.c1 AS c_4, 1 AS c_6, ref_1.c3 AS c_9
   FROM (t3 ref_0 RIGHT JOIN t1 ref_1
          ON (ref_0.c_0 = ref_1.c2));
INSERT INTO t1 VALUES (11000, 0, null, 0, 0.0, 15);
--- Original query, result set: {0}
SELECT COUNT(*) AS c_6
FROM (t1 AS ref_0 LEFT OUTER JOIN t4 AS ref_1
       ON (ref_0.c0 = ref_1.c_2))
WHERE ref_1.c_3 =
    DCBRT(CASE WHEN ref_0.c2 LIKE '7%z'
               THEN ref_1.c_6 ELSE ref_0.c4 END);
--- Transformed query, result set: {1}
SELECT COUNT(*) AS c_6
FROM (t1 AS ref_0 LEFT OUTER JOIN t4 AS ref_1
       ON (ref_0.c0 = ref_1.c_2))
WHERE (CASE WHEN ((ref_1.c_9 >= ref_1.c_4)
            OR (NOT (ref_1.c_9 >= ref_1.c_4))
            OR ((ref_1.c_9 >= ref_1.c_4) IS NULL))
       THEN ref_1.c_3 ELSE ref_1.c_3 END) =
    DCBRT(CASE WHEN ref_0.c2 LIKE '7%z'
               THEN ref_1.c_6 ELSE ref_0.c4 END);
```

Figure 9: PostgreSQL logic bug triggered by both the original query and the transformed query.

Example 5: Bug triggered in both two queries. Figure 9 shows the test case where both the original query and the transformed query trigger the same logic bug in PostgreSQL. EET transforms the expression **ref_1.c_3** in the **WHERE** clause of the original query to a **CASE WHEN** expression, which unexpectedly makes the transformed query return different results. We reported the test case to PostgreSQL developers, who confirmed that both the two queries in the test case triggered a bug according to their query plans. Figure 10 shows their query plans. Their query plans are nearly the same, and the only difference is that the original query used a hash join, while the transformed query used a hash right join. Both query plans are problematic because they lose join filters, which are responsible for filtering the rows that satisfy the

```
QUERY PLAN: Original Query / Transformed Query
Aggregate
|-- Hash Join / Hash Right Join
      Hash Cond: (ref_1.c5 = ref_0.c0)
MISSING: Join Filter: (ref_1.c4 / CASE... = dcbrt(...))
   |-- Nested Loop Left Join
         Join Filter: ('1'::text = ref_1.c2)
      |-- Seq Scan on t1 ref_1
      |-- Materialize
         |-- Seq Scan on t2 ref_1_1
               Filter: (c1 > c1)
   |-- Hash
      |-- Seq Scan on t1 ref_0
```

Figure 10: Query plans of the original query (using hash join) and the transformed query (using hash right join) in Figure 9.

predicate in the **WHERE** clauses. The root cause of this logic bug is that PostgreSQL removed some unnecessary **LEFT JOIN** tables that are underneath other **LEFT JOIN** but failed to clean the data structures referencing the removed tables, causing PostgreSQL to consider that the join filters affected by such data structures are unreasonable and drop them. After the bug is fixed, the query plans of both the origin query and the transformed query contain their corresponding join filters.

5.4 Comparative Study

To check whether EET can find logic bugs missed by existing approaches, we investigate the earliest bug-inducing versions of the 35 logic bugs EET found and check whether these versions are before the existing approaches got published. We would conclude that EET can find logic bugs missed by existing approaches if some long-latent bugs are found by EET. This comparison is reasonable and objective because: (1) all logic bugs found by EET are not marked as duplicates by developers, meaning that no approach found these bugs until EET found them; (2) all the 5 DBMSs in our evaluation have been extensively tested by existing approaches [4, 11, 25–27, 29, 35], meaning that the existing approaches did not find the long-latent bugs found by EET during their evaluation.

We classify the existing approaches according to the years they got published, resulting in 2 classes: 2020 (PQS [27], TLP [26], NoREC [25]) and 2023 (TQS [35], Pinolo [11], DQE [29]). Therefore, we check whether the versions inducing the logic bugs found by EET are before 2020 and 2023. Table 6 shows the results.

Among 35 logic bugs found by EET, 13 already existed before 2020 (*i.e.*, bugs were involved in 2019 or earlier), indicating that all the existing approaches miss these 13 logic bugs in their extensive evaluation, as all these approaches are proposed in or after 2020. In addition, 11 logic bugs can be triggered between 2020 and 2022, while none of the three approaches (*i.e.*, TQS, Pinolo, and DQE) published in 2023 found these bugs. These results indicate that existing approaches indeed miss many long-latent logic bugs. Due to the inherent limitations of query-level manipulation, these approaches have to limit the patterns of their generated queries and thus cannot be applied to complex queries (*e.g.*, the bug-triggering queries in Figure 2), while many DBMS bugs can be triggered only by complex queries [13]. Empowered by expression-level manipulation, EET can be easily applied to complex queries and thus successfully found many logic bugs missed by existing approaches.

Table 6: Latency of the logic bugs found by EET

DBMS	Found	Bug-involved year		Longest latency
		< 2023	< 2020	
MySQL	10	9	6	6 years
PostgreSQL	3	1	1	20 years
SQLite	9	5	4	8 years
ClickHouse	11	7	2	4 years
TiDB	2	2	0	3 years
Total	35	24	13	20 years

We investigate the longest latency of the logic bugs found by EET for each tested DBMS. The result is shown in Table 6. Interestingly, for each tested DBMS, EET can find at least one bug whose latency is longer than 3 years. The bug with the longest latency, *i.e.*, 20 years, is found in PostgreSQL, which is shown in Figure 2. These results demonstrate that EET can effectively find long-latent bugs.

While EET can find many logic bugs missed by existing approaches, it may inherently miss some bugs that can be found by existing approaches. For example, if a logic bug makes both the original query and the transformed query produce the same incorrect results, EET will miss this bug. However, approaches like PQS [27] can help detect such missed bugs by inferring the expected query results. One interesting future work is to integrate EET and existing approaches into a testing framework that can efficiently schedule these approaches during testing DBMSs.

6 Related Work

Logic Bug Detection in DBMSs. Several approaches have been proposed to detect logic bugs in DBMSs [11, 25–27, 29, 35]. PQS [27] synthesizes a query that guarantees to return a specific row using its manually implemented interpreter. If the tested DBMS fails to fetch the row, PQS identifies a logic bug. NoREC [25] generates a query with a predicate and transforms this query by moving its predicates from its **WHERE** clause to its **SELECT** clause. NoREC identifies a logic bug if the moved predicate produces different results. TLP [26] partitions an original query into three separated queries by decomposing its predicate. The union of the re-

sults of these separated queries should be consistent with the result of the original one, otherwise, a logic bug is triggered. SQLancer [30] integrates the above three techniques and has been deployed to test various DBMSs. In addition, DQE [29] generates different types of queries (*e.g.*, **`SELECT`**, **`UPDATE`**, and **`DELETE`**) with the same predicates. These different queries should operate the same rows, otherwise, a logic bug is triggered. Pinolo [11] modifies the predicate of a query, making its constraints looser/stricter. Therefore, the modified query should return a superset/subset of the results of the original query, otherwise, Pinolo identifies a logic bug. TQS [35] applies schema normalization [7, 22] to slit a wide table into multiple tables and construct a customized query using join tables, whose ground truth results can be inferred from the original wide table. Based on these ground truth results, TQS finds many logic bugs related to **`JOIN`** operations.

As discussed in Section 1 and 2, all these approaches are based on *query-level manipulation*, which requires the approaches to understand the semantics of the manipulated queries. Therefore, existing approaches cannot be applied to complex queries whose semantics are complicated. Different from existing approaches, EET is based on *expression-level manipulation*, which operates on expressions and has no necessity to understand the query semantics. Therefore, EET does not need to limit the query patterns and thus can be applied to various queries.

DBMS Test-Case Generation. Without focusing on logic bugs, some approaches [9, 13, 17, 33, 38] are proposed to generate more diverse test cases for DBMSs. SQLsmith [33] is a grammar-based DBMS fuzzer, which embeds the AST rules of SQL language and can generate complex SQL statements. SQUIRREL [38] proposes a new intermediate representation to model SQL queries and infers the dependencies between statements. In this way, SQUIRREL can generate queries that contain multiple SQL statements. Griffin [9] performs grammar-free mutation to test DBMSs by summarizing the DBMS state information into its metadata graph and mutating SQL queries according to the graph to prevent semantic errors. DynSQL [13] performs dynamic query interaction to capture the latest DBMS state information and incrementally generate complex and valid queries. In addition, some approaches [2, 18] are proposed to improve the test-case generation for logic bug detection. SQLRight [18] enables code coverage feedback, which gives the supported test oracles more chance to find logic bugs in rarely executed DBMS code. QPG [2] records the covered query plans during testing DBMSs and prioritizes mutating the queries that trigger new query plans, which is more likely to expose a new logic bug.

These approaches and EET can complement each other. On one hand, the high-quality and diverse queries generated by these approaches can help EET find more logic bugs hidden in the corner logic of DBMSs. On the other hand, the general test oracle provided by EET can enable these approaches to catch more various bugs.

7 Conclusion

In this paper, we propose a new and fine-grained methodology, *expression-level manipulation*, for approaching logic-bug-detection problems in DBMSs without limiting the query patterns. Based on this methodology, we propose a novel and general approach, *equivalent expression transformation (EET)*, which can effectively find logic bugs using two transformations: determined boolean expressions and redundant branch structures. We evaluate EET on 5 widely used DBMSs. In total, EET found 66 bugs, 35 of which are logic bugs. Many of these logic bugs have long latency and are missed by existing approaches. We believe the generality and effectiveness of EET can inspire more follow-up research on DBMS testing.

Acknowledgments

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

References

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

[2] Jinsheng Ba and Manuel Rigger. Testing database engines via query plan guidance. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, pages 2060–2071, 2023.

[3] ClickHouse. https://clickhouse.com/.

[4] Using SQLancer to test ClickHoouse and other DBMS. https://presentations.clickhouse.com/?path=heisenbug2021.

[5] Irving M Copi, Carl Cohen, and Victor Rodych. *Introduction to logic*. Routledge, 2018.

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

[7] Jim Diederich and Jack Milton. New methods and fast algorithms for database normalization. *ACM Transactions on Database Systems (TODS)*, 13(3):339–365, 1988.

[8] Daniela Florescu, Alon Y. Levy, and Alberto O. Mendelzon. Database techniques for the world-wide web: A survey. In *Proceedings of the 1998 International Conference on Management of Data (SIGMOD)*, pages 59–74, 1998.

[9] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. Griffin: Grammar-free dbms fuzzing. In *Proceedings of the 37th International Conference on Automated Software Engineering (ASE)*, pages 1–12, 2022.

[10] Github. https://github.com/.

[11] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. Pinolo: Detecting logical bugs in database management systems with approximate query synthesis. In Julia Lawall and Dan Williams, editors, *Proceedings of the 2023 USENIX Annual Technical Conference (ATC)*, pages 345–358, 2023.

[12] Jan L Harrington. *Relational database design and implementation*. Morgan Kaufmann, 2016.

[13] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. DynSQL: Stateful fuzzing for database management systems with complex and valid sql query generation. In *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[14] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. Detecting transactional bugs in database engines via graph-based oracle construction. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 397–417, 2023.

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

[16] Raghav Kaushik and Ravi Ramamurthy. Efficient auditing for complex sql queries. In *Proceedings of the 2011 International Conference on Management of Data (SIGMOD)*, pages 697–708, 2011.

[17] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. Sequence-oriented dbms fuzzing. In *Proceedings of the 2023 International Conference on Data Engineering (ICDE)*, 2023.

[18] Yu Liang, Song Liu, and Hong Hu. Detecting logical bugs of DBMS with coverage-based guidance. In *Proceedings of the 31st USENIX Security Symposium*, pages 4309–4326, 2022.

[19] Elliott Mendelson. *Introduction to mathematical logic*. CRC press, 2009.

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

[21] Oracle Database. https://www.oracle.com/database/.

[22] Thorsten Papenbrock and Felix Naumann. Data-driven schema normalization. In Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß, editors, *Proceedings of the 20th Internationa Conference on Extending Database Technology (EDBT)*, pages 342–353, 2017.

[23] PostgreSQL contributor gifts. https://wiki.postgresql.org/wiki/Contributor_Gifts.

[24] PostgreSQL. https://www.postgresql.org/.

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

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

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

[28] Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Özcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. Athena++ natural language querying for complex nested sql queries. *Proceedings of the 46th International Conference on Very Large Databases (VLDB)*, pages 2747–2759, 2020.

[29] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. Testing database systems via differential query execution. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, pages 2072–2084, 2023.

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

[31] DBMS bugs found by SQLancer. https://www.manuelrigger.at/dbms-bugs/.

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

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

[34] SQL standard. https://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

[35] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. Detecting logic bugs of join optimizations in dbms. In *Proceedings of the 2023 International Conference on Management of Data (SIGMOD)*, pages 1–26, 2023.

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

[37] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex sql queries using automatic summary tables. In *Proceedings of the 2000 International Conference on Management of Data (SIGMOD)*, pages 105–116, 2000.

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

[39] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 34(3):1096–1116, 2020.