Detecting Logic Bugs in DBMSs via Equivalent Data Construction

WENQIAN DENG, Tsinghua University, China
JIE LIANG*, Beihang University, China
ZHIYONG WU, Tsinghua University, China
JINGZHOU FU, Tsinghua University, China
YU JIANG*, Tsinghua University, China

Database Management Systems (DBMS) perform various data operations such as arithmetic calculations and string manipulations when executing SQL queries. These operations are complex due to the wide range of data types and the intricate interactions between different data. Consequently, errors in implementing these data operations can lead to logic bugs, potentially causing issues such as implicit type coercion, overflow, and precision loss. Existing logic bug detection methods primarily focus on issues introduced during query optimization by adapting query-level strategies. However, these methods have limitations when it comes to detecting logic bugs caused by implementation errors in data types and operations.

To address this, we propose *equivalent data construction* (EDC), a novel approach to detect logic bugs in data operation implementations within DBMSs. The core insight is that for data operation expressions in SQL queries, substituting them with precomputed result values should yield identical query outcomes. EDC mainly involves the following steps: first, it constructs equivalent data for a given operation by computing and storing the results in a transformed table; then, it transforms the query by replacing the operation expressions with the corresponding precomputed results. Any inconsistencies between the results of the base and transformed queries indicate potential logic bugs. We implemented EDC and evaluated it on seven well-tested and widely-used DBMSs(e.g., MySQL, MariaDB). Our evaluation revealed 54 previously unknown bugs, of which 39 have been confirmed by developers. The findings prompted active engagement from developers. For example, MariaDB developers described our findings as counterintuitive, helping them uncover more issues related to data operations.

CCS Concepts: • **Information systems → Data management systems**.

Additional Key Words and Phrases: Database, Logic bug

ACM Reference Format:
Wenqian Deng, Jie Liang, Zhiyong Wu, Jingzhou Fu, and Yu Jiang. 2025. Detecting Logic Bugs in DBMSs via Equivalent Data Construction. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 314 (December 2025), 25 pages. https://doi.org/10.1145/3769779

1 Introduction

Database Management Systems (DBMSs) contain various data operations, such as arithmetic calculations and string manipulations [35]. These operations are inherently complex due to the diverse range of data types involved, as well as the need to handle boundary conditions and precise

*Jie Liang and Yu Jiang are the corresponding authors.

Authors' Contact Information: Wenqian Deng, Tsinghua University, Beijing, China; Jie Liang, Beihang University, Beijing, China; Zhiyong Wu, Tsinghua University, Beijing, China; Jingzhou Fu, Tsinghua University, Beijing, China; Yu Jiang, Tsinghua University, Beijing, China.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2836-6573/2025/12-ART314
https://doi.org/10.1145/3769779

control over data representation. For example, arithmetic operations on floating-point numbers require careful management of rounding errors, while string manipulations must account for varying character encodings and length constraints. Moreover, the complexity of data operations also arises from the interactions between these operations and the underlying data types, which are influenced by system architecture, configuration, and optimization rules. For instance, data operations may involve implicit type conversions and varying treatments of different data types, as well as handling implicit coercions, type overflows, or precision loss. Thus, logic bugs related to data operations in DBMSs are more likely to arise not from individual operations in isolation, but from subtle inconsistencies that emerge when multiple operations and data types interact in complex ways. This complexity makes the implementation of data operations inherently error-prone.

There have been several studies on detecting logic bugs in DBMSs. For instance, EET [27] targets expression-level transformations, TLP [37] partitions queries based on predicate logic, and NoREC [36] rewrites queries to disable DBMS optimizations. Radar [45] introduces metadata constraints to find potential bugs. While effective for exposing query planner and optimizer issues, such as incorrect rewrites, joins, or predicate logic, previous tools mainly focus on the structural aspects of query logic. They are not primarily designed to detect issues that stem from how data operations behave over different data types, such as errors caused by implicit type coercions, overflows, or precision loss. In this paper, we focus on a different class of correctness issues: logic bugs arising from data types and operations. These issues often occur during the execution of data operation expressions and are orthogonal to the logic targeted by previous tools.

The main challenge for detecting such logic bugs arising from complex and diverse data operations lies in the lack of ground truth results. Data operations, such as arithmetic calculations and string manipulations, often involve intricate interactions between different data types. These interactions become even more complex when boundary conditions and unusual input patterns are introduced, further increasing the likelihood of subtle processing errors. In the absence of reference outputs, it becomes difficult to determine whether such behaviors are semantically correct, especially when the intended behavior depends on subtle and undocumented implementation details. Therefore, these issues are particularly challenging to identify, as they rarely result in obvious query-level bugs but instead manifest as inconsistencies in how data is processed, transformed, or compared.

To address this, we propose **Equivalent Data Construction** (EDC), a novel approach to detect logic bugs in data operation implementations within DBMSs. EDC is specifically designed to target logic bugs arising from data types and operations, filling a significant gap in prior techniques that have less explored this space. The core insight is that *for data operation expressions in SQL queries, replacing them with precomputed results should yield identical query results.* It has the following steps: ① Randomly select a data operation and generate a base table as the input. ② Compute the results of the operation expression based on the data in the base table and store the results in a newly transformed table. Note that the table is equivalent to the base table with respect to the operation expression. ③ Generate a base `SELECT` query on the base table. ④ Generate a transformed query by replacing operation expressions (e.g., `CONCAT`$(a1, a2)$) in the base query with the corresponding precomputed data (e.g., r) in the transformed table. ⑤ Compare the results of the query pairs, and if they are inconsistent, it indicates a potential bug.

Figure 1 illustrates our approach on a test case that triggers a logic bug in MariaDB. This bug had a widespread impact, affecting the use of the `CONCAT` operation with data containing strings, numbers, dates, and other types. We generate this test case by constructing data equivalence based on the `CONCAT` operation. Specifically, we first select the `CONCAT` operation and generate a base table containing a `VARCHAR` column and a `TEXT` column populated with a single row of data. Next, we compute the results of the expression `CONCAT`$(a1, a2)$ based on the data in the base table and store these results in the transformed table as r. After that, we first generate a base `SELECT` query

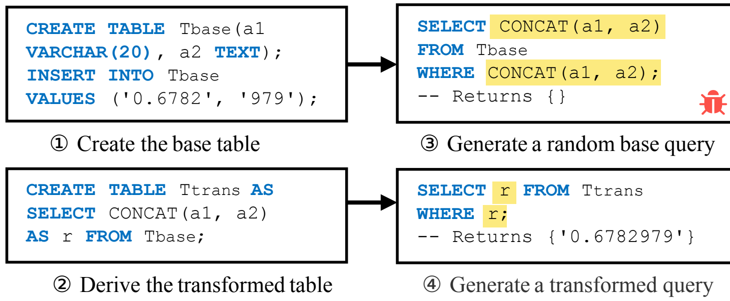

Fig. 1. Illustrative example, based on a detected bug by EDC in MariaDB.

on the base table using the expression CONCAT($a1, a2$) in the WHERE condition. Then, we create a transformed query by replacing the expression CONCAT($a1, a2$) with the precomputed result r from the transformed table. When we execute the two queries, the base query incorrectly returns zero rows despite a valid match, whereas the transformed query correctly returns one. This discrepancy reveals a bug in how MariaDB handles the CONCAT operation under this condition.

When we reported this bug to the developers, they identified the root cause as follows: "*In some contexts, text strings containing numbers where the integer part is zero and the fractional part is non-zero were erroneously evaluated as* FALSE *in* WHERE *conditions.*" [2] They further discovered that the bug was important and needed to be fixed because it affected multiple data types and involved changes to 10 internal classes, requiring modifications to more than a thousand lines of code. Our approach is effective in detecting this kind of logic bug because it innovatively establishes metamorphic relationships by replacing data with its equivalents. This enables the exploration of a broad range of combinations involving different data types and operations, facilitating the detection of data-level issues in DBMSs. In contrast, existing approaches often emphasize query-level equivalence, addressing a different aspect of DBMSs.

To evaluate the effectiveness of EDC, we used it to test seven widely-used DBMSs, i.e., MySQL [10], MariaDB [8], Percona [4], PostgreSQL [13], TiDB [15], OceanBase [12] and ClickHouse [5]. Although these databases had undergone thorough testing, EDC still reported a total of 54 bugs. Among these, 39 bugs have been confirmed by the developers of the corresponding DBMSs. The reported issues received active attention from the developer communities. For example, TiDB developers actively inquired about our approach and acknowledged its effectiveness, while MariaDB developers described our findings as counterintuitive, helping them uncover more issues related to data operations. We also evaluate EDC against 3 state-of-art testing techniques, i.e., TLP [37], Radar [45], and EET [27]. In 24-hour experiments, EDC detected 38 bugs across seven DBMSs, including 35 that were not reported by existing tools. This result underscores the distinct bug-detection scope of EDC and its potential to complement prior approaches. We make the following contributions:

- We demonstrate that logic bugs arising from data operations, such as type conversions and precision errors, can be significant. However, these issues are often underrepresented in the findings of existing tools, which tend to focus primarily on query-level transformations.
- We propose a novel approach EDC that is designed to detect DBMS logic bugs in data operations. EDC constructs equivalent data by computing the results of operation expressions from the base table to derive a transformed table. EDC executes queries on both tables and identifies discrepancies, indicating potential bugs.
- We implemented EDC and used it to test seven widely-used DBMSs. It uncovered 54 previously unknown bugs, with 39 of these bugs confirmed by the developers.

2 Background and Motivation

2.1 Data Types and Operations

Data Types. In DBMSs, data types define the kind of data that can be stored in a database column, ensuring consistency, precision, and efficiency in data storage and retrieval [14]. They can be categorized into four major groups: (1) Numeric types, which encompass integers, floating-point numbers, etc., are essential for arithmetic operations and numerical computations; (2) String types, which store text-based and binary data, include fixed-length types (e.g., `CHAR`), variable-length types (e.g., `VARCHAR`), and binary types (e.g., `BLOB`) for handling large or unstructured data such as multimedia files; (3) Date types, which manage temporal data such as `DATE`, `TIME`, and `TIMESTAMP`, enable precise tracking and manipulation of time-based information, making them indispensable for scheduling, event logging, and analytics; and (4) Custom types, which extend the functionality of DBMS by supporting specialized formats like `JSON` data types and spatial data types(e.g., `POINT`).

Data Operation. DBMSs offer a variety of operators and functions; in this paper, we collectively refer to them as data operations. These operations enhance the capability of DBMSs to execute complex SQL queries and facilitate in-depth data analysis [1, 3]. We focus on the following four categories of operations: (1) Arithmetic operations, which perform element-wise computations on numeric columns, such as addition (+) and subtraction (−), are essential for numerical transformations; (2) Function operations, which leverage built-in or user-defined functions to process data, such as string manipulations (e.g., `CONCAT`) and date-time calculations (e.g., `DATE_ADD`), enhance the versatility of SQL queries; (3) Predicate operations, which evaluate conditions or comparisons between columns, returning boolean results, such as comparisons (>, <, =), null checks (`IS NULL`), and set-based operations (`IN`, `BETWEEN`), are critical for filtering; and (4) Aggregate operations, which summarize data across multiple rows, producing grouped or single-value results through functions like `SUM`, `AVG`, and `MAX`.

The diversity of data types and operations serves as the cornerstone of a DBMS's ability to support a broad spectrum of functionalities. By leveraging this combination, DBMSs effectively address diverse application requirements while ensuring performance, reliability, and user-friendliness.

2.2 Limitations of Existing Approaches

Several approaches have been proposed to detect logic bugs in DBMSs [27, 36–38, 44, 45], many of which operate at the query level and rely on various strategies to identify inconsistencies in query results. Specifically, some approaches split or transform queries to expose discrepancies in the results (e.g., TLP [37], NoREC [36]), while others construct logically crafted SQL queries designed to return specific results in a controlled manner (e.g., PQS [38], DQE [44]). While query-level strategies have proven effective in many cases, their focus on high-level semantics or syntactic transformations means they may not specifically target the types of subtle bugs that can emerge from complex data operations or type interactions. Additionally, some techniques (e.g., EET [27]) apply rule-based expression rewriting, while others, such as Radar [45] focus on detecting metadata-related logic bugs by comparing results between a database with metadata constraints and a raw database. While these methods do not rely directly on data semantics, they operate at a different level of abstraction and are not primarily intended to capture data relationships.

Notably, the bug illustrated in Figure 1 triggered by a query containing `WHERE CONCAT`$(a1, a2)$ is uniquely detected by our approach. EET operates at the syntactic level, and while its rule-based expression rewrites provide valuable insights, they may not fully capture deeper data relationships. NoREC aims to reveal bugs by transforming query semantics, but its approach is primarily focused on a specific pattern of predicate rewriting, which may limit its coverage. Radar, which emphasizes metadata constraints, offers useful perspectives but does not manipulate data expressions directly,

potentially missing deeper inconsistencies in data computation. Taken together, these approaches do not explicitly model equivalence across data types and operations, which may make it challenging to detect certain types of data-level logic bugs.

2.3 Definitions of Data Equivalence

To illustrate our idea, we first present formal definitions of the relevant concepts, based on relational algebra expressions [33]. Table 1 summarizes the used relational algebra notation.

Table 1. Relational algebra notation.

Symbol	Meaning
Π	Projection: selects a subset of attributes from a relation
ρ	Renaming: assigns new names to a relation or its attributes
$\mathcal{G}$	Grouping aggregation: computes over grouped tuples

Data Equivalence. Let T be a table with columns $A_1, A_2, \ldots, A_n$, and let op be a data operation (either a row-wise computation or a group-wise aggregation). We define a transformed table T' to be *data-equivalent* to T under operation op, denoted as:

$$T' \overset{op}{\equiv} T$$

if and only if one of the following holds:

- **Row-wise Equivalence.** If op is a row-wise operation applied to a subset of columns in T, then:
$$T' = \rho_{T'(A_q,\ldots,A_n,R)} \left(\Pi_{A_q,\ldots,A_n,\, op(A_1,\ldots,A_p)} (T) \right)$$
This means we first project columns $A_q, \ldots, A_n$ and the computed result of $op(A_1, \ldots, A_p)$, then rename the result to obtain table T' with a new result column R.
- **Group-wise Equivalence.** If op is a group-wise aggregation over group-by attributes $A_q, \ldots, A_n$, then:
$$T' = \rho_{T'(R_q,\ldots,R_n,R)} \left({}_{A_q,\ldots,A_n}\mathcal{G}_{op(A_1,\ldots,A_p)} (T) \right)$$
Here, we group T by attributes $A_q, \ldots, A_n$, apply the aggregation op to $A_1, \ldots, A_p$, and then rename the result table and attributes accordingly.

In both cases, T' is derived from T using a deterministic operation op, and $\overset{op}{\equiv}$ captures the equivalence between original and transformed data under that operation. We refer to T as the *base table*, T' as the *transformed table*, and $op(A_1, \ldots, A_p)$ as the *equal operation expression*.

Basic Idea of EDC. Let T be the base table and T' be the transformed table such that:

$$T' \overset{op}{\equiv} T$$

Let Q be a SQL query over T that includes occurrences of $op(A_1, \ldots, A_p)$. We define a rewritten query Q' by replacing each such expression with the precomputed column R from T':

$$Q' = Q \left(op(A_1, \ldots, A_p) \ \mapsto \ R \right)$$

This guarantees the equivalence of query results:

$$T' \overset{op}{\equiv} T \ \Rightarrow \ Q(T) \equiv Q'(T')$$

Our insight is that replacing data operation expressions (i.e., $op(A_1, \ldots, A_p)$) with their precomputed results (i.e., R) in SQL queries should produce identical outputs. This principle

ensures that the semantics of a query remain unchanged, regardless of whether the expression is computed dynamically or retrieved from a precomputed column. By leveraging this, we can systematically generate equivalent SQL queries: one relying on operation expressions and the other using precomputed results. These pairs enable rigorous testing of DBMS behaviors, revealing inconsistencies. Compared to existing techniques, this form of data equivalence checking provides an orthogonal and complementary perspective on testing. Indeed, this approach led to the discovery of the logic bug shown in Figure 1, where substituting a `CONCAT` expression with its precomputed value exposed a violation of expected equivalence.

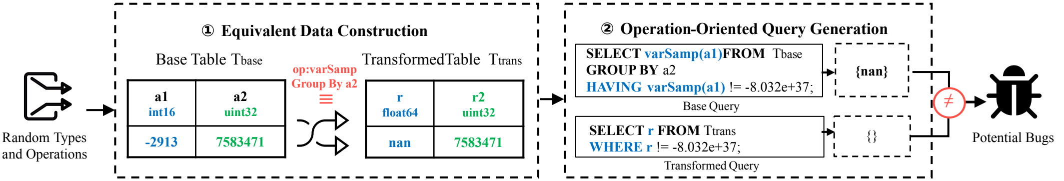

Fig. 2. The overview of EDC. It consists of two stages: Equivalent Data Construction (Section 3.2) and Operation-Oriented Query Generation (Section 3.3). In the first stage, a base table is generated as input for an operation, and the results are computed and stored in a derived transformed table, ensuring data equivalence on the operation. In the second stage, random queries on the base table are transformed into queries on the transformed table, with result inconsistencies revealing bugs.

3 Design of EDC

We propose Equivalent Data Construction (EDC), a novel technique designed to detect logic bugs in DBMSs that stem from data operation implementations. The core idea of EDC is that for data operation expressions in SQL queries, replacing them with precomputed results should yield identical query results. If executing the original query and its transformed counterpart yields different results, EDC identifies this discrepancy as a potential logic bug in the underlying DBMS.

3.1 Overview of EDC

Figure 2 provides an overview of EDC, using the bug shown in Figure 9 as a running example to illustrate its detection workflow. The process is structured into two stages: equivalent data construction (Section 3.2), and operation-oriented query generation (Section 3.3). In stage ①, we randomly select a data operation (e.g., `varSamp`) and some data types (e.g., `int16` and `int32`) and generate a base table based on the selected operation and types. Next, we insert random data for the base table and compute the results of the equal operation expression based on the operation, storing the results in a newly derived transformed table. For example, as shown in Figure 2, the operation `varSamp` is applied to $a1$, grouped by $a2$ in the base table, and the result value (e.g., NaN for the group where $a2$ equals 7583471) is stored in the transformed table r.

In stage ②, semantically equivalent SQL queries are generated to detect potential bugs in the DBMS data operations handling by validating the data equivalence relationship between the base table and the transformed table. First, a random query is constructed on the base table by referencing the equal operation expression `varSamp`$(a1)$. Then, it is transformed into a query by replacing all occurrences of the equal operation expression with the result column r. If the equivalent expression forms a condition within a `HAVING` clause, this condition is moved to the `WHERE` clause in the transformed query. Finally, to identify logic bugs in the DBMS, the results of each query are checked for consistency. If the results are different, we consider the test case to find a potential bug related to the data operation.

3.2 Equivalent Data Construction

In this section, we discuss how to construct the base table, denoted as T_{base}, and the transformed table, denoted as T_{trans}, while ensuring data equivalence between them.

3.2.1 Base Table Generation. We begin by selecting a random data operation op (e.g., +, `CONCAT`, `AVG`), collected from the list of supported operations detailed in the target DBMS documentation. We explicitly exclude operations known to produce non-deterministic results, such as `ANY_VALUE`. These operations may return varying outputs, and thus are unsuitable for detecting logic bugs based on consistent data behavior. Next, we construct a base table T_{base} with a configurable number of columns, each assigned a randomly selected data type supported by the DBMS. Before populating the table, we perform a compatibility check to ensure that the target operation op is syntactically and semantically valid for the selected column types. This validation is conducted by issuing a lightweight `SELECT` query that applies op to the candidate column types. If the query executes successfully without producing type errors or semantic violations, the type-operation combination is deemed valid. Conversely, if the DBMS returns an error indicating type incompatibility, the current configuration is discarded, and a new combination of data types and an operation is generated. This process is repeated until a valid and executable setup is identified.

Once the compatibility of the type-operation combination is confirmed, we proceed to populate T_{base} with a batch of randomly generated data. This data serves as the foundation for further processing, including the construction of its transformed table T_{trans} and the evaluation of data equivalence in subsequent query transformations. To ensure both general coverage and bug-triggering potential, EDC employs a structured data generation strategy that accounts for a wide range of type-specific behaviors and edge cases. For each column, values are generated in accordance with the type's constraints to ensure validity. This prevents common runtime errors such as `INSERT` failures due to out-of-range or malformed values. To increase the likelihood of exposing type-related bugs, EDC introduces some cross-type data patterns, where values from one logical domain are deliberately inserted into fields of a different type. For instance, a text column may contain numeric-looking strings (e.g., `"123"`, `"-45.6"`), allowing tests to exercise implicit type coercion during operations such as addition and concatenation. Similarly, for unsigned integer fields, EDC may insert negative values or values exceeding the type's precision, triggering truncation or overflow behavior during query execution. These hybrid and invalid inputs help uncover inconsistencies in how DBMSs internally handle type-related operations. In addition, EDC emphasizes the generation of boundary and edge-case values, including extremely large or small numbers, empty strings, and mixed-case text, which are known to stress the expression evaluation logic.

3.2.2 Transformed Table Generation. After the data generation for the base table T_{base}, the corresponding results are computed based on the selected equal operation and stored in a derived transformed table (i.e., T_{trans}). Figure 3 illustrates the process of transformed table generation. First, we use the target DBMS's type inference capabilities to determine the type of the equal operation expression ($OpExpr$, e.g., `CONCAT`$(a1, a2)$). The inferred type, obtained from the system table, is essential for constructing the operation-transformed table to store the computed results. To obtain this type, we begin by creating a view over the base table that projects the equal operation expression. We then query the system catalog to retrieve the inferred type of the expression within this view. Finally, using the retrieved $type$ information, we construct the transformed table T_{trans} with the query `CREATE TABLE` T_{trans} `(`r $type$`)`. For DBMSs that support the `CREATE TABLE AS` syntax (e.g., MySQL), the process can be further streamlined. In such cases, the transformed table can be created using the statement `CREATE TABLE` T_{trans} `AS (SELECT (`$OpExpr$`) AS` r `FROM`

T_{base}). This simplifies the workflow by combining table creation and data computation. If an error occurs during table generation, EDC skips the current test case and continues to the next.

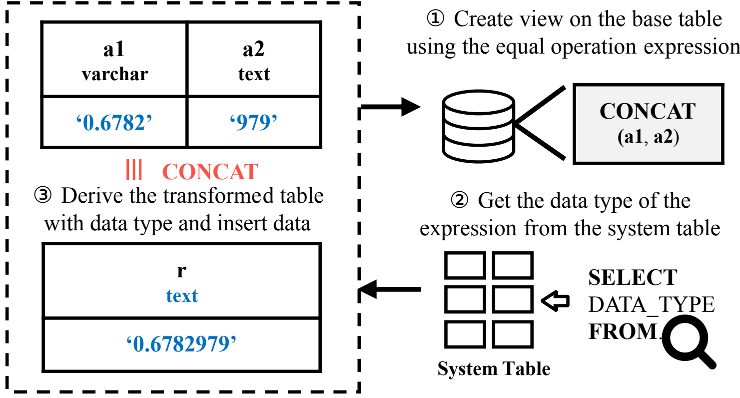

Fig. 3. The process of transformed table generation.

After constructing the transformed table, we compute and insert data into it by evaluating the selected operation expression over the base table, ensuring data equivalence between the two tables. The specific process depends on whether the selected expression is non-aggregate or aggregate:

(1) Non-Aggregate Operations: For non-aggregate operations, such as +, `CONCAT`, or `NULLIF`, the operation is applied row by row on the selected columns in T_{base}. For example, the operation expression `CONCAT`($a1$, $a2$) in T_{base} generates a single column r in T_{trans}, as shown in Figure 3. The data is then inserted into T_{trans} using a direct query `INSERT INTO` T_{trans} `(SELECT CONCAT(`$a1$, $a2$`) AS` r `FROM` T_{base}`)`. To increase query diversity, T_{base} may include additional columns that are not involved in the tested operation expression. These columns are copied unchanged into T_{trans}, preserving structural consistency between the two tables while maintaining the correctness of the equivalence transformation. These additional columns enable richer query construction by participating in predicates, projections, or function arguments. Such interactions increase the syntactic and semantic complexity of generated queries, which can expose subtle logic bugs. For example, as shown in Figure 7, TiDB exhibited a failure when a JSON column ($a2$, an additional column) was compared with a boolean expression ($a1$ IS NULL) in a `WHERE` clause, illustrating how these auxiliary columns can contribute to bug detection.

(2) Aggregate Operations: For aggregate operations such as `SUM` or `AVG`, the process involves grouping data in T_{base} based on one or more grouping columns. The aggregate function is then applied within each group to compute the results. The transformed table T_{trans} stores both the grouping columns and the corresponding aggregated values. For example, the following query populates T_{trans} with two columns $r2$ and r: `INSERT INTO` T_{trans} `(SELECT` $a2$ `AS` $r2$`, SUM(`$a1$`) AS` r `FROM` T_{base} `GROUP BY` $a2$`)`. To ensure that the aggregated results preserve data equivalence, it is important that T_{base} contains only the grouping columns and those involved in the aggregation. This contrasts with non-aggregate operations, where unrelated columns can be retained without affecting correctness.

3.3 Operation-Oriented Query Generation

After constructing the base table T_{base} and the transformed table T_{trans}, EDC proceeds to generate a batch of `SELECT` query pairs that operate on the two tables and are expected to yield equivalent results. The overall process is described in Algorithm 1, with expression generation details

elaborated in Algorithm 2. The purpose of Algorithm 1 is to construct a query pair (Q_{base}, Q_{trans}) that is logically equivalent. It first constructs a base query Q_{base} over T_{base} by applying randomly generated expressions to its columns (Lines 1–9). It then transforms Q_{base} into Q_{trans} by replacing sub-expressions that correspond to previously materialized operations with their precomputed results in T_{trans} (Line 11). The purpose of Algorithm 2 is to synthesize valid expressions for use within Algorithm 1. It incrementally constructs valid expressions by selecting node types, such as constants, columns, and operators, and combining them according to syntactic and type constraints.

Algorithm 1: Operation-Oriented Queries Generation

```
   Input: Tables T_base, T_trans, Selected Data Operation op
   Output: Query Q_base, Q_trans
 1 OpExpr ← composeCol(T_base.InputCols, op);
 2 Q_base ← Select(T_base);
 3 if op is aggregate then
 4 |   Q_base.addWhere(genExpr(T_base.GroupCols, 0));
 5 |   Q_base.addGroupBy(T_base.GroupCols);
 6 |   Q_base.addHaving(genExpr({OpExpr}, 0));
 7 else
 8 |   Q_base.addWhere(genExpr({OpExpr} ∪
 9 |    T_base.OtherCols, 0));
10 end
11 Q_trans ← Transform Q_base to query on T_trans via defined rules;
12 return Q_base, Q_trans
```

Specifically, Algorithm 1 begins by composing the target operation expression *OpExpr* from the selected operation *op* and its input columns (Line 1). This expression (e.g. CONCAT($a1, a2$)) is treated as a conceptual column, as it represents a unit of computation whose result is materialized in T_{trans}. Treating *OpExpr* as a column also facilitates query transformation, where it is directly replaced by its precomputed column during the construction of Q_{trans}. Then, a base query Q_{base} is initialized as a simple SELECT over the base table T_{base} (Line 2). The rest of the query structure depends on whether the selected operation is aggregate or non-aggregate: For aggregate operations, a WHERE clause is first added using a randomly generated predicate over the grouping columns ($T_{base}.GroupCols$, Line 4). A GROUP BY clause is then added using these same columns (Line 5), followed by a HAVING clause that filters based on a predicate over the composed *OpExpr* (Line 6). For non-aggregate operations, a single WHERE clause is added based on a predicate involving both *OpExpr* and the unrelated columns in $T_{base}.OtherCols$ (Lines 8–9).

The next step in Algorithm 1 is query transformation (Line 11), where the transformed query Q_{trans} is derived from the base query Q_{base} by rewriting expressions according to a set of predefined rules summarized in Table 2. For non-aggregate operations, each occurrence of *OpExpr* (i.e., $op(A_1, \ldots, A_p)$) in Q_{base} is replaced with the corresponding precomputed result column R from the transformed table T_{trans}. For aggregate operations, the transformation replaces both the aggregate expression and the GROUP BY clause with their equivalent columns in T_{trans}. Specifically, the aggregate is replaced with column R, and each grouping attribute $A_q, \ldots, A_n$ is mapped to columns $R_q, \ldots, R_n$. EDC applies WHERE filters only to columns in the GROUP BY clause. This is because, after aggregation, the raw values of non-grouped columns are no longer available in the transformed table, making post-aggregation filtering on those columns infeasible. For example, in SELECT-GROUP_CONCAT(a, b) FROM t GROUP BY c, applying a filter like WHERE a > 10 becomes infeasible

on the equivalent table because columns a and b have already been aggregated and their raw values are unavailable for comparison. Additionally, any `HAVING` clause that references the aggregate expression is rewritten as a `WHERE` clause over R. Finally, by executing queries and comparing the results of Q_{base} and Q_{trans}, any discrepancies indicate potential logic bugs in the DBMS, such as issues with expression evaluation, type conversion, aggregate computations, or query optimizations.

Lines 4–6 and 8 of Algorithm 1 invoke the expression generation procedure `genExpr`, which is responsible for constructing randomized SQL predicates over the relevant columns. This procedure is detailed in Algorithm 2. It takes as input a set of columns and two depth parameters: `CurDepth`, the current depth of the expression tree, and `MaxDepth`, a fixed configuration constant that limits recursion and prevents overly deep expression trees. At each invocation, the algorithm first initializes a candidate set of node types, starting with leaf-level types: constants and column references (i.e., `CONST` and `COLUMN`, Line 2). If the current depth is less than the maximum allowed, internal node types, such as logical and comparison types, are also added to the candidate set (Lines 3–5). These node types are collected from DBMS documentation. A node type is then randomly selected from the candidate set. If the selected type is a leaf node, the algorithm calls `randomValueNode`, which returns either a constant value (e.g., `42`, `'abc'`) or a randomly selected column from the input set, depending on the node type (Lines 7–8). For non-leaf nodes, a new operator node is initialized, and its required number of children (e.g., two for `AND` or `+`) is recursively generated by invoking `genExpr` with increased depth (Lines 10–13).

Algorithm 2: Expression Generation

```
Input: Columns Cols, Current depth of query CurDepth, and Max depth MaxDepth
Output: Generated Expression Node
1  Function genExpr(Cols, CurDepth):
2      NodeTypes ← {CONST, COLUMN};
3      if Depth < MaxDepth then
4          NodeTypes ← NodeTypes ∪ {AND_EXPR, OR_EXPR, ...}, which enabling logical
             composition of expression nodes;
5      end
6      Type ← random(NodeTypes);
7      if Type ∈ {CONST, COLUMN} then
8          Node ← randomValueNode(Cols, Type)
9      else
10         Node ← initNode(Type);
11         for i from 1 to Node.ChildNum
12          Node.addChild(genExpr(Cols, CurDepth + 1));
13     end
14 return Node
```

4 Implementation

We implement EDC in Python, supporting seven DBMSs: MySQL, MariaDB, Percona, PostgreSQL, TiDB, OceanBase, and ClickHouse. The structure and details of the codebase are available on the website of EDC [6]. To prepare for testing, we extracted supported data types and operations from the official documentation [9, 11] and built-in test cases. Specifically, we collected 381, 387, 381, 1351, 287, 382, and 953 operations from MySQL, MariaDB, Percona, PostgreSQL, TiDB, OceanBase,

Table 2. The basic rules for transforming queries

Type	Data Equivalence Relationship	Query Transformation Rule
Non-Aggregate	$T' = \rho_{T'(A_q,\dots,A_n,R)}(\Pi_{A_q,\dots,A_n,op(A_1,\dots,A_p)}(T))$	**SELECT** $\text{expr}(A_q,\dots,A_n,\text{op}(A_1,\dots,A_p))$ **FROM** T_{base} **WHERE** $\text{expr}(A_q,\dots,A_n,\text{op}(A_1,\dots,A_p)) \rightarrow$ **SELECT** $\text{expr}(A_q,\dots,A_n,R)$ **FROM** T_{trans} **WHERE** $\text{expr}(A_q,\dots,A_n,R)$
Aggregate	$T' = \rho_{T'(R_q,\dots,R_n,R)}\left({}_{A_q,\dots,A_n}\mathcal{G}_{op(A_1,\dots,A_p)}(T)\right)$	**SELECT** $\text{expr}(A_q,\dots,A_n,\text{op}(A_1,\dots,A_p))$ **FROM** T_{base} **WHERE** $\text{expr}(A_q,\dots,A_n)$ **GROUP BY** $A_q,\dots,A_n$ **HAVING** $\text{expr}(op(A_1,\dots,A_p)) \rightarrow$ **SELECT** $\text{expr}(R_q,\dots,R_n,R)$ **FROM** T_{trans} **WHERE** $\text{expr}(R_q,\dots,R_n)$ **AND** $\text{expr}(R)$

The notation $expr(col)$ represents an expression generated using col as a basic node in the expression tree.

and ClickHouse, respectively, covering 36 distinct data types. The implementation of EDC generally comprises two steps:

Test Case Generation. In the equivalent data construction stage, we generate random diverse values, including boundary cases, null values, and mixed data, to explore potential edge cases and maximize test coverage. Boundary cases are specifically selected based on DBMS documentation [7]. For numeric and date types, we use minimum/maximum allowed values (e.g., `TINYINT` ranges from -128 to 127), while for string and custom types, we include edge cases like empty strings and `NULL` values. The maximum number of rows inserted into the table is configurable, with a default setting of 100 rows. The number of columns in the table is also configurable, with a maximum of 5 input columns for the selected data operations in our implementation. For non-aggregate operations, up to 3 additional unrelated columns are included to increase query complexity. For aggregate operations, up to 3 grouping columns are added to support `GROUP BY` operations. After the equivalent data construction stage, no further DML operations are performed on the original or derived transformed tables to ensure data consistency throughout the testing process.

In the operation-oriented query generation stage, we generate `SELECT` query pairs for the base table and the transformed table. The maximum number of `SELECT` queries generated per test is configurable, with a default of 1,000. We generate and transform queries by leveraging the Abstract Syntax Tree (AST) representation. During iterative traversal of the AST, tree nodes corresponding to the equal operation expression are identified and replaced with the equivalent column node from the derived transformed table. This transformation ensures that the semantics of the queries remain consistent across both tables.

Bug Identification. EDC identifies logic bugs by executing the base and equivalent queries on the original and transformed tables and comparing results. If any inconsistency is detected during execution, the test is terminated early to focus on analyzing the discrepancy. Before comparison, EDC sorts the query results to ensure reliability, including sorting list objects within individual rows. When both queries return error messages, we ignore these cases in our analysis as they do not indicate logic bugs. This decision was based on initial feedback from developers who indicated that such outcomes were considered consistent behavior since both queries failed. When the results of the two queries differ, EDC flags the case as a potential bug. These discrepancies are logged with detailed information, including the DBMS information, the specific queries executed, the results obtained, and the nature of the inconsistency.

5 Evaluation

In this section, we evaluate the effectiveness of EDC in detecting logic bugs. We begin by presenting previously unknown logic bugs discovered by EDC and highlighting their severity. Next, we analyze

the distribution of bugs across data types and data operations to show EDC's ability to handle diverse scenarios, from simple to complex types and operations. We also examine key statistics, including the distribution of test cases, column counts, and the ratio of executed queries to unique bugs identified. In addition, we include case studies to further highlight EDC's effectiveness, showing how it can enhance existing testing methods by revealing additional bugs. Finally, we compare EDC with other related DBMS testing techniques to demonstrate how data equivalence can complement existing approaches.

5.1 Evaluation Setup

Tested DBMSs. To evaluate the generality and efficiency of EDC, we select seven popular open-source DBMSs, namely MySQL [10, 49], MariaDB [8, 17], Percona [4], PostgreSQL [13], TiDB [24, 34], OceanBase [12], and ClickHouse [5], which are widely used in industry. Table 3 shows the specific versions of the tested DBMSs for the experiments. The type in Table 3 follows the prior work [45] and established classifications [25]. Specifically, we applied EDC to MySQL 9.0.1, MariaDB 11.4.3, Percona 8.4.2, PostgreSQL 16.1, TiDB 8.2, OceanBase 4.2.1, and ClickHouse 24.9. These DBMSs have been extensively tested by many existing approaches [21, 27, 29, 30, 38, 50]. However, existing tools have recently identified significantly fewer bugs in these DBMSs. Consequently, discovering new bugs in these DBMSs has become highly challenging.

Table 3. Target DBMSs: version, complexity, popularity, and diversity overview.

DBMS	Version	LOC	GitHub Stars	Type
MySQL	9.0.1	6.2M	10.8k	Relational
MariaDB	11.4.3	2.3M	5.6k	Relational
Percona	8.4.2	5.0M	1.2k	Relational
PostgreSQL	16.1	1.9M	18k	Relational
TiDB	8.2	1.5M	36k	NewSQL
OceanBase	4.2.1	7.5M	8.5k	Distributed
ClickHouse	24.9.1	2.2M	36k	Analytical

Experimental Setup. We perform the experiments on a machine running 64-bit Ubuntu 20.04 with 128 cores (AMD EPYC 7742 Processor @ 2.25 GHz) and 504 GiB of main memory. To detect real-world logic bugs, we perform testing of EDC on all seven DBMSs continuously. For quantitative comparisons, all DBMSs tested are run in Docker containers that can be downloaded from their website with 5 CPU cores and 32 GiB of main memory for 24 hours.

5.2 Detected Logic Bugs

Bug Statistics. Table 4 shows the statistics of logic bugs reported by EDC for each DBMS. EDC generated a total of 24,132 valid type-operation combinations for testing. Using these combinations, EDC discovered 54 logic bugs across seven widely used and well-tested DBMSs, specifically 13, 10, 11, 2, 5, 6, and 7 logic bugs in MySQL, MariaDB, Percona, PostgreSQL, TiDB, OceanBase, and ClickHouse, respectively. Among them, 39 anomalies have been confirmed as previously unknown bugs. Additionally, 4 bugs are considered duplicates of existing ones, and 11 anomalies remain under investigation due to their complexity. To distinguish unique bugs, EDC applies a post-analysis de-duplication process based on query structure, data type, and operation patterns. Final confirmation is conducted in collaboration with DBMS developers when necessary. Further details on the de-duplication strategy are provided in Section 6. The results reflect that data behavior related logic

Table 4. Number and status of previously unknown bugs detected by EDC

DBMS	Reported	Confirmed	Duplicate	Investigating
MySQL	13	11	2	0
MariaDB	10	3	2	5
Percona	11	11	0	0
PostgreSQL	2	1	0	1
TiDB	5	5	0	0
OceanBase	6	6	0	0
ClickHouse	7	2	0	5
Total	54	39	4	11

bugs are prevalent in these popular DBMSs. With the equivalent data construction approach, EDC can trigger various data operations behaviors and detect these bugs.

Bug Severity. We analyze the severity of the logic bugs reported across various DBMSs, highlighting their substantial impact on system reliability and correctness. These bugs have a wide impact because they involve fundamental data operations that could be invoked by various higher-level expressions in DBMSs. Among the reported bugs, we found that 12 bugs were classified as critical or major due to their potential to compromise query results or system stability. Specifically, type conversion errors and precision loss during arithmetic operations were often marked as important by developers, as they directly affect data integrity and query accuracy.

Developers of various DBMSs acknowledged the practical value of identifying these bugs. For example, TiDB developers have actively inquired about our approach and praised its effectiveness, and MariaDB developers described our findings as counterintuitive, helping them uncover more issues related to functions and data types, recognizing the operational value of EDC beyond academic novelty. These findings not only resolved immediate issues but also contributed to improving the overall robustness of the DBMSs.

5.3 Comprehensive Bug Analysis

We analyzed the 54 bugs identified to highlight the data types and operations that EDC can effectively cover. This analysis demonstrates the broad capability of EDC in detecting issues across diverse data types and its effectiveness in handling various data operations. Then, we further investigate the characteristics of test cases that are more likely to expose logic bugs, including their data types and query structural features. Through this comprehensive analysis, we aim to provide insights into both EDC's detection capabilities and potential strategies for future testing.

Bug Distribution in Data Types. We investigated the bugs detected by EDC and analyzed the primary data types involved. The results are presented in Table 5. For each bug, we examine the query that triggers inconsistent behavior to identify the related data types. When a query contains interactions between multiple data types, all main participating types are included in our analysis. As shown, 4 data type categories (i.e., numeric, string, date, and custom type) are all involved in the 54 bugs identified. Specifically, 31 bugs are related to numeric types, 9 to string types, 13 to date types, and 10 to custom types.

Numeric type bugs dominate, making up 57.4% of the total, due to implicit conversions and issues with unsigned and floating-point types. Date type bugs account for 24.1% of the total, primarily from improper handling of date functions and comparisons between dates and numeric types. String type bugs, which represent 16.7%, often occur during conversions from strings to numeric types. 3 of custom-type bugs are related to `JSON`, particularly in `JSON`-related functions and predicates.

Table 5. Bug distribution across different data types.

DBMS	Numeric	String	Date	Custom
MySQL	6	2	5	3
MariaDB	5	4	0	3
Percona	5	1	4	2
PostgreSQL	2	0	1	0
TiDB	4	1	1	1
OceanBase	3	1	1	1
ClickHouse	6	0	1	0
Total	31	9	13	10

Table 6. Bug distribution across different data operations.

DBMS	Arithmetic	Function	Predicate	Aggregate
MySQL	1	10	1	1
MariaDB	2	8	0	0
Percona	2	8	1	0
PostgreSQL	0	1	0	1
TiDB	0	3	1	1
OceanBase	1	3	2	0
ClickHouse	0	4	1	2
Total	**6**	**37**	**6**	**5**

The remaining 7 bugs are associated with spatial types, primarily caused by incorrect handling of geographic data in functions and conditional evaluations.

Bug Distribution in Data Operations. Table 6 lists the data operation types (i.e., arithmetic, function, predicate, and aggregate operation) in the 54 test cases triggering the logic bugs. Among the 54 identified bugs, 37 were caused by function operations, 6 by arithmetic operations, 6 by predicate operations, and 5 by aggregate operations. Function-related bugs appear in all seven DBMSs and represent the largest category in our findings. This is likely because functions are the most diverse operation type, supporting a wide range of operations across various column data types. The majority of predicate-related bugs, which account for 6 cases, are likely caused by improper handling of logical expressions or conditional evaluations, such as those in `WHERE` clauses. Arithmetic-related bugs, also accounting for 6 cases, typically stem from incorrect type conversions or edge cases during mathematical evaluations. They show that even basic operations may have implementation errors and can fail under specific conditions. Aggregate-related bugs account for 4 cases. These bugs primarily stem from two sources: incorrect handling of `HAVING` conditions and improper processing of empty results during aggregation operations.

Test Case Characteristics and Bug-Prone Patterns. To further evaluate our testing approach, we first provide a high-level overview of the test case corpus and its relationship to bug discovery. As shown in Table 7, we evaluated a total of 24,132 test cases, each generating over 100 queries, resulting in approximately 2.41 million executed queries. These efforts led to the discovery of 54 unique logic bugs, yielding an average of 44,689 queries (or 447 test cases) per unique bug. We then analyzed the composition of test cases by data type. The overall distribution consisted of numeric (8,236 cases, 34%), string (10,348 cases, 43%), date (2,801 cases, 12%), and custom types (2,747 cases, 11%). Interestingly, the bug distribution across these types, shown in Table 5, does not mirror their

Table 7. Distribution of test cases by data type.

Type	Numeric	String	Date	Custom
Test Cases(%)	8,236 (34%)	10,348 (43%)	2,801 (12%)	2,747 (11%)

representation in the test set. For example, although numeric types accounted for only 34% of the test cases, they were responsible for 57.4% of the detected bugs. This discrepancy may suggest that numeric inputs may be more sensitive to subtle logic issues. In addition to data types, we investigated the structural complexity of the queries that triggered bugs, focusing on the number of columns involved. Based on the minimal form of queries that revealed each bug, we found that most were relatively simple in structure: 2 bugs were triggered by single-column queries, 18 by two-column queries, 30 by three-column queries, and 4 by four-column queries. While queries with five or more columns were included in the test set, the queries that ultimately triggered bugs involved fewer columns.

5.4 Selected Bugs Found by EDC

To better illustrate the categories of these bugs, we present a selection of confirmed examples showcasing the diverse and complex issues uncovered by EDC. Specifically, we selected six representative bugs that span different data types and operation types across the seven DBMSs. To emphasize the core issues, the examples have been simplified to isolate the root cause. Instead of presenting original test cases, we distilled the queries and data to their minimal form, removing unrelated elements to eliminate unnecessary complexity while preserving the essence of each bug.

Case 1: Inconsistent numeric handling in MySQL function operation COALESCE. Figure 4 demonstrates an inconsistency in how MySQL evaluates the result of the COALESCE function when handling numeric type columns. The absence of a WHERE clause in this case is due to simplification during test reduction, which resulted in the predicate being removed. The base table T_{base} contains two BIT columns, $a1$ and $a2$, where $a1$ is set to NULL and $a2$ is set to 1. The transformed table T_{trans} is created by applying the COALESCE function to the two columns, resulting in an equal column r. When querying the base table directly, the result of COALESCE($a1$, $a2$) is represented as 0x31, which is the ASCII encoding of the numeric value 1. However, when querying the derived transformed table T_{trans}, (R1.M1)where the result is stored as r, the output is correctly interpreted as numeric 1.

```
-- Set up the base table
CREATE TABLE Tbase (a1 BIT, a2 BIT);
INSERT INTO Tbase VALUES (NULL, 1);
-- Set up the transformed table
CREATE TABLE Ttrans AS SELECT COALESCE(a1, a2) AS r FROM Ttrans;
-- Base query, returns {0x31}
SELECT COALESCE(a1, a2) FROM Tbase;
-- Transformed query, returns {1}
SELECT r FROM Ttrans;
```

Fig. 4. Inconsistent numeric handling in MySQL function operation COALESCE.

This inconsistency arises from how MySQL handles BIT data types in different contexts. In the base query, MySQL treats the result of COALESCE as a binary form of string. This is likely because BIT values may be implicitly encoded as strings when returned as the output of the function COALESCE. This results in the hexadecimal output 0x31, which is the binary representation of the

number 1 in ASCII. In the derived transformed table, MySQL stores the result as a numeric value, which ensures the transformed query returns the correct numeric value.

Case 2: Inconsistent evaluation of DATE_ADD function in MySQL with date type. Figure 5 illustrates a discrepancy in MySQL's evaluation of DATE_ADD when used dynamically in a BETWEEN clause. In this case, the base table T_{base} contains two columns: $a1$ as DATETIME and $a2$ as INT. Then, the base query dynamically evaluates the result of DATE_ADD within the BETWEEN clause to compare it against a range. Despite the operation being within the specified range, MySQL fails to return any rows in the base query. In contrast, when the result is precomputed and stored as column r in the derived transformed table T_{trans}, the transformed query successfully evaluates the range condition and returns the correct result. The root cause of this inconsistency lies in how MySQL handles dynamic function evaluations in conditions like BETWEEN. When DATE_ADD is used within a BETWEEN clause, MySQL struggles to evaluate it correctly when comparing it against constants. This leads to no rows being returned, even though the result should fall within the range. In contrast, when the DATE_ADD result is stored in the derived transformed table, MySQL processes it correctly.

```
-- Set up the base table
CREATE TABLE Tbase (a1 DATETIME, a2 INT);
INSERT INTO Tbase VALUES ('6722-01-03 04:01:57', '3525');
-- Set up the transformed table
CREATE TABLE Ttrans AS SELECT DATE_ADD(a1, INTERVAL a2 DAY) AS r FROM Tbase;
-- Base query,returns {}
SELECT DATE_ADD(a1, INTERVAL a2 DAY) FROM Tbase
WHERE DATE_ADD(a1, INTERVAL a2 DAY)
BETWEEN -3.18812e+38 AND '2935-07-28 18:13:52';
-- Transformed query, returns {6731-08-29 04:01:57}
SELECT r FROM Ttrans WHERE r
BETWEEN -3.18812e+38 AND '2935-07-28 18:13:52';
```

Fig. 5. Inconsistent evaluation of DATE_ADD function in MySQL with date type.

Case 3: Inconsistent handling in Percona ST_Collect function with spatial types. Figure 6 illustrates an inconsistency in Percona's evaluation of the spatial function ST_Collect when used within a WHERE clause. In this case, the base table T_{base} contains a single column $a1$ of type POLYGON, populated with a distinct polygon. The base query applies ST_Collect(c0) directly within a subquery in the WHERE clause and checks for membership of a given LINESTRING using the IN operator. Then, this query returns a single row with NULL as the result of the SELECT clause. In contrast, a semantically equivalent query is executed on T_{trans}. When the same LINESTRING is tested for membership against this precomputed value, the query returns an empty result set, with no rows matched. The root cause of this inconsistency lies in how Percona evaluates spatial aggregation functions like ST_Collect in dynamic versus materialized contexts. When the function is used directly in a query over the base table, the internal handling may result in ambiguous evaluation, causing the output to be NULL. However, once the result is materialized into a new table, the evaluation context changes, leading to a complete and consistent representation of the geometry collection, resulting in no match being found at all.

Case 4: Inconsistent handling of JSON data type and predicate operation in TiDB. Figure 7 involves an inconsistency in TiDB when evaluating comparisons between JSON data and boolean expressions in a query. The base table T_{base} contains three columns: $a1$ of type DOUBLE, $a2$ of type JSON, and $a3$ of type BIGINT. The derived transformed table T_{trans} contains a column r generated from the

```
-- Set up the base table
CREATE TABLE Tbase(a1 POLYGON);
INSERT INTO Tbase VALUES(ST_GeomFromText(
'POLYGON(61.85 74.8, 78.08 26.70, 96.77 77.69)'));
-- Set up the transformed table
CREATE TABLE Ttrans AS SELECT ST_Collect(a1) AS r FROM Tori;
-- Base query, returns {NULL}
SELECT ST_Collect(a1) FROM Tbase
WHERE ST_GeomFromText('LINESTRING(-21 -45, 154 -24)')
IN (SELECT ST_Collect(a1) FROM Tbase);
-- Transformed query, returns {}
SELECT r FROM Ttrans
WHERE ST_GeomFromText('LINESTRING(-21 -45, 154 -24)')
IN (SELECT r FROM Ttrans);
```

Fig. 6. Inconsistent handling of spatial data type and operation in Percona.

expression $a1$ IS NULL. Then, the base query performs a comparison between the JSON column $a2$ and the boolean expression a1 IS NULL in the WHERE clause. The transformed query compares $a2$ to r. However, the transformed query returns an empty result set. The root cause of this inconsistency is TiDB's improper handling of implicit type conversion when comparing JSON data types with boolean expressions. Instead of converting the JSON value to a numeric type, TiDB performs an inconsistent implicit cast, leading to erroneous results.

```
-- Set up the base table
CREATE TABLE Tbase (a1 DOUBLE, a2 JSON, a3 BIGINT);
INSERT INTO Tbase VALUES (-2.790e+28, 'false', 229109314);
-- Set up the transformed table
CREATE TABLE Ttrans (r BIGINT, a2 JSON, a3 BIGINT);
INSERT INTO Ttrans SELECT (a1 IS NULL), a2, a3 FROM Tori;
-- Base query, returns {false, 229109314}
SELECT a2, a3 FROM Tbase WHERE a2 <= (a1 IS NULL) ORDER BY a2 ASC, a3 ASC;
-- Transformed query, returns {}
SELECT a2, a3 FROM Ttrans WHERE a2 <= r ORDER BY a2 ASC, a3 ASC;
```

Fig. 7. Inconsistent handling of JSON data type and predicate operation in TiDB.

Case 5: Record duplication in TiDB aggregate operation with string type. Figure 8 demonstrates a discrepancy in TiDB when executing the aggregate operation AVG. The base table T_{base} contains a float column $a1$ and a binary column $a2$. Then, the derived transformed table T_{trans} has two columns, representing which were returned from executing AVG($a1$) GROUP BY $a2$ on the base table. When querying the base table, the query produces two correct records with distinct values. However, querying the derived transformed table unexpectedly returns duplicate records, with each original record appearing twice.

The root cause lies in TiDB's query optimization process, where the IN subquery in the transformed query is rewritten as an INNER JOIN for performance improvement. During this process, type conversions are applied to VARBINARY values in $r2$. This leads to a loss of distinctness, causing duplicate records to be returned by the transformed query, while the base query remains correct.

Case 6: Inconsistent handling of NaN values in ClickHouse aggregate operation with numeric type. Figure 9 shows an inconsistency in ClickHouse's handling of NaN values in the aggregate operation varSamp. The base table T_{base} contains two numeric columns. Then, the derived transformed table

```
-- Set up the base table
CREATE TABLE Tbase (a1 FLOAT, a2 VARBINARY(11));
INSERT INTO Tbase VALUES (-3.21411e+37, 0x22693D9BBE4);
INSERT INTO Tbase VALUES (4.53528e+37, 0x2BA66016016);
-- Set up the transformed table
CREATE TABLE Ttrans (r DOUBLE, r2 VARBINARY(11));
INSERT INTO Ttrans SELECT AVG(a1), a2 FROM Tbase GROUP BY a2;
-- Base query,
   returns{-8.039000231889152e+36, \aQ\xf7\xd9*\xf2}
          {8.415999773927147e+37, \vя\xf9\xda\x18}
SELECT AVG(a1), a2 FROM Tbase WHERE (a2 OR a2) IN
(SELECT a2 FROM Tbase WHERE a2 <= 0x99) GROUP BY a2 HAVING AVG(a1);
-- Transformed query,
   returns {-8.039000231889152e+36, \aQ\xf7\xd9*\xf2}
           {-8.039000231889152e+36, \aQ\xf7\xd9*\xf2}
           {8.415999773927147e+37, \vя\xf9\xda\x18}
           {8.415999773927147e+37, \vя\xf9\xda\x18}
SELECT r, r2 FROM Ttrans WHERE (r2 OR r2)
IN (SELECT r2 FROM Ttrans WHERE r2 <= 0x99) AND r;
```

Fig. 8. Record duplication in TiDB aggregate operation with string type.

T_{trans} is created with two columns: r, representing the result of varSamp($a1$) GROUP BY $a2$, and $r2$, which is carried over from the GROUP BY result. A query on the base table with a HAVING clause filtering rows where varSamp($a1$) != -8.032e+37 returns NaN, 7583471. However, the transformed query on the derived transformed table, which checks the condition r != -8.032e+37, returns an empty result.

The root cause of this bug lies in how ClickHouse handles NaN values in tables that define an ORDER BY key. When a column containing NaN values is included in the ORDER BY clause, ClickHouse adopts a special treatment of these values during query execution. Unlike regular numeric values, NaNs are considered unordered and do not follow standard comparison semantics. As a result, ClickHouse excludes rows containing NaN values from certain comparison operations, most notably those involving inequality, such as !=. This behavior causes such rows to be silently omitted from the result set, even when they logically should be included. Consequently, this can lead to incomplete or misleading query results, especially in analytical scenarios where the presence of NaN values is significant.

```
-- Set up the base table
CREATE TABLE Tbase (a1 INT16, a2 UINT32) ORDER BY a1;
INSERT INTO Tbase VALUES (-2913, 7583471);
-- Set up the transformed table
CREATE TABLE Ttrans ORDER BY r AS
SELECT varSamp(a1) AS r, a2 AS r2 FROM Tori GROUP BY a2;
-- Base query, returns {nan, 7583471}
SELECT varSamp(a1), a2 FROM Tbase GROUP BY a2
HAVING varSamp(a1) != -8.032e+37;
-- Transformed query, returns {}
SELECT r, r2 FROM Ttrans WHERE r != -8.032e+37;
```

Fig. 9. Inconsistent handling of NaN values in ClickHouse aggregate operation with numeric type.

Why these bugs are only detected by EDC? EDC is designed to detect logic bugs by leveraging a test oracle based on equivalent data construction (EDC). This approach focuses on verifying the correctness of internal data operations by replacing expressions with their precomputed equivalents and checking for inconsistencies in results. Unlike methods that primarily operate at the query level, such as EET [27], or those that rely on metadata constraints like Radar [45], EDC evaluates data-level behavior directly, offering a complementary strategy to existing techniques.

Additionally, our analysis of the bugs identified by EDC reveals several key factors that contribute to its effectiveness. First, it systematically exercises rarely tested combinations of operations and data types, increasing coverage of less conventional execution paths. Second, EDC evaluates a wide range of operation invocations and type conversions, enabling it to uncover edge cases that previous approaches fail to consider. Third, its use of equivalence-based transformations can influence query optimization paths, thereby exposing execution-level bugs that are otherwise hidden from conventional testing strategies. Through its data equivalence strategy, EDC is able to generate rich and diverse test cases that penetrate deep into the internal logic of DBMSs, ultimately enabling the discovery of subtle and previously unreported bugs.

5.5 Comparison with Existing Techniques

Compared Tools. We compare EDC against 3 state-of-the-art open-source tools for finding logic bugs in DBMSs: TLP [37], EET [27], and Radar [45]. TLP [37] generates randomized databases and SQL queries to uncover incorrect query results, flagging logic bugs when expected rows are not retrieved. EET [27] detects logic bugs by performing semantics-preserving transformations on complex SQL queries to ensure the transformed queries yield the same results as the originals. Additionally, Radar [45] identifies logic bugs by comparing query results on databases with identical data but differing metadata constraints, such as `NOT NULL`. We ran the testing tools on each target DBMS for 24 hours and recorded the number of detected logic bugs as the metric. For TLP and Radar, we conducted experiments using the default configurations for each DBMS. For EET, we used the `-ignore-crash` option to bypass crash bugs, allowing EET to focus solely on detecting logic bugs. This setting ensures a fair comparison, as EDC is also designed specifically to target logic bugs rather than crashes.

Results. Table 8 displays the number of logic bugs detected by each tool in 24 hours. During the evaluation, EDC found 8, 6, 6, 2, 5, 6, and 7 bugs in MySQL, MariaDB, Percona, PostgreSQL, TiDB, OceanBase, and ClickHouse, respectively, while TLP detected 3 bugs in MySQL, 2 in Percona, 4 in TiDB, and none in others. Radar identified 1 bug in MySQL and Percona, 5 in TiDB, with no support for ClickHouse, OceanBase, and PostgreSQL, and no bugs detected in MariaDB. EET found 3 bugs in MySQL, 1 in PostgreSQL, no bugs in TiDB, and 2 in ClickHouse, but it does not support MariaDB, Percona, and OceanBase. Across the seven DBMSs, EDC detected 38 unique bugs, while TLP, Radar, and EET identified 9, 7, and 5 bugs respectively, highlighting the complementary nature of their testing strategies.

Furthermore, to better understand the relationship between the bugs, we investigated the bug reports generated by these 4 tools. Figure 10 provides insight into the relationships between the bugs detected by EDC and the other tools. In our evaluation, EDC uniquely identified 40 bugs, demonstrating its capability to uncover a broad range of issues. Among these, 3 bugs overlap with TLP, while no overlap exists with Radar or EET.

Analysis. EDC and TLP share 3 overlapping bugs, all related to SQL functions. This is because both tools utilize SQL functions in their testing methodologies. However, EDC detects significantly more bugs related to SQL functions and extends its coverage to other data operations, such as predicate operations, which TLP might miss. The 6 unique bugs detected by TLP involve `JOIN` operations, which are outside the scope of EDC, as it focuses on single-table queries. The bugs detected

Table 8. Number of detected logic bugs in 24 hours. "-" indicates that the tool does not support the DBMS.

DBMS	EDC	TLP	Radar	EET
MySQL	8	3	1	3
MariaDB	6	-	0	-
Percona	6	2	1	-
PostgreSQL	2	0	-	1
TiDB	5	4	5	0
OceanBase	6	0	-	-
ClickHouse	7	0	-	2
Total	40	9	7	6

by EDC and EET are entirely orthogonal. EET specializes in finding bugs associated with advanced SQL constructs like `WITH` clauses and window functions. In contrast, EDC focuses on exploring the diversity of data types and data operations, uncovering bugs related to type conversions, and data handling. Similarly, the bugs detected by EDC and Radar are also orthogonal. Radar identifies metadata-related logic bugs by creating databases with varying metadata constraints. In contrast, EDC does not involve metadata changes but focuses on equivalence relationships to detect bugs.

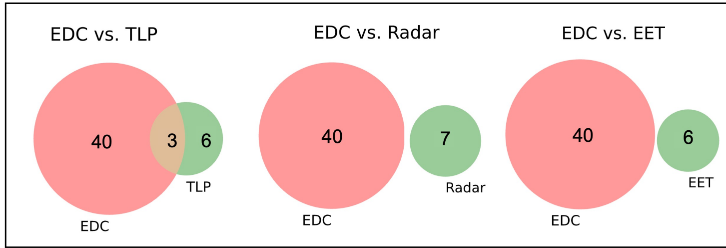

Fig. 10. Venn diagram for bugs found by different tools.

6 Discussion

Effort of Adaptation. EDC is designed for easy adaptation to new DBMSs. The main integration effort involves analyzing the target DBMS's documentation to extract supported data types and operations, which are then configured into EDC to enable automatic generation of system-specific equivalence relationships. To further support customization, EDC provides extensible interfaces for random SQL query generation, allowing developers to incorporate DBMS-specific features, such as proprietary syntax, unique aggregate functions, or engine-specific behaviors. Importantly, EDC is not bound by differences in SQL dialects or interpretations of the SQL standard. It defines correctness based on a database-agnostic invariant: replacing a data operation with its precomputed equivalent should preserve query results, regardless of syntax or engine semantics. This focus on logical equivalence at the data level allows EDC to generalize across diverse platforms.

Current Constraints and Extensibility. While EDC has demonstrated effectiveness across a wide range of DBMSs, its support for certain advanced SQL features remains a work in progress. One such limitation involves window functions like `RANK`, which are challenging to support due to their order-sensitive semantics. They could potentially be handled through advanced transformation

techniques, such as rewriting window functions into semantically equivalent forms using self-joins or subqueries. Another ongoing extension involves support for experimental data types, such as VECTOR in TiDB and MariaDB. Supporting these types primarily involves adapting EDC's syntax parser and query generation components to accommodate their usage in expressions.

Input Diversity and Test Space Coverage. EDC systematically explores a diverse test space along two dimensions: input data and query structure. At the data level, each test case is populated with up to 100 rows by default, containing varied values such as boundary cases (e.g., minimum/-maximum integers and dates), NULLs, and coercion-prone inputs. Value generation is stratified by data type to ensure balanced coverage across numeric, string, date, and other system-specific types. At the query level, up to 1,000 SELECT queries are generated per test using randomized expression trees (Algorithm 2). While generation is randomized, it is guided by type constraints, operator sets (e.g., AND, <, CASE), and a configurable depth limit. This design enables exploration of a wide range of expression structures while maintaining semantic validity.

Bug Deduplication of EDC. During our continuous testing phase, EDC uncovered over 800 issues across different DBMSs. However, a substantial portion of these were duplicate reports stemming from the same underlying problem. Simply counting all reported issues would therefore lead to a significant overestimation of the actual number of unique bugs. To ensure an accurate assessment, we conducted a systematic deduplication process by grouping issues that shared the same root causes, data type interactions, or operation-level anomalies. This allowed us to identify and report truly distinct logic bugs with greater precision.

In detail, the deduplication process involved three steps. First, we conducted root cause analysis by inspecting the related data types and operations of query execution paths to differentiate distinct bugs from recurring manifestations of the same defect. Second, we grouped issues based on their affected operations and data types to identify overlaps; for example, multiple test cases exposing the same arithmetic overflow for a specific data type were merged into a single representative bug. Finally, each unique issue was manually reviewed to ensure it represented a distinct logic error rather than a variation of an existing one. Through this deduplication process, we reduced the initial issues to 54 unique bugs, providing a precise and realistic assessment of EDC 's effectiveness.

False Positives and False Negatives. The false positives detected by EDC primarily stem from inconsistencies or behaviors that developers consider intentional or outside the scope of bugs. During EDC 's experiments, 4 bug reports were marked as "won't fix" by the developers and subsequently closed. Although these cases exhibited inconsistent behavior, the developers did not regard them as bugs. For instance, one reported case in MySQL involved inconsistent results with the AVG function, which developers clarified as expected behavior since such usage was unsupported outside of GROUP BY queries, leading to a "won't fix" classification. As for false negatives, like most testing-based techniques, EDC does not aim for complete bug coverage. Its detection capabilities are scoped to inconsistencies that arise from violations of data equivalence. To mitigate the risk of false negatives within its target domain, EDC generates a wide range of operation-type combinations and expression structures to increase coverage. EDC can serve as a complementary component within a broader DBMS testing framework. By combining EDC with approaches that emphasize query semantics or constraint-based inference, users can achieve more comprehensive DBMS testing.

7 Related Work

DBMS Logic Bug Detection. Many works have proposed various methods for detecting logic bugs in DBMSs [21, 23, 27, 36, 37, 39, 43–46]. NoREC [36] works by generating a query with a predicate and then transforming the query to one that DBMSs cannot optimize. A logic bug is detected if the output changes after this transformation. TLP [37] breaks down the original query

into three separate subqueries by splitting the predicate. The combined results of these subqueries should align with the original query's result; if not, a logic bug is reported. PQS [39] generates queries that are designed to always return a specific row by utilizing a custom interpreter. If the DBMS under test fails to retrieve this row, PQS identifies it as a logic bug. Sedar [21] summarizes a series of SQL function patterns and generates test cases based on that. The core idea of DQE [44] is to detect logic bugs of DBMSs by comparing whether different SQL queries with the same predicate access the same rows in the database. TQS [46] decomposes wide tables into smaller ones and generates custom queries with join operations, using the base table to derive ground truth results. PINOLO [23] synthesizes queries that produce supersets or subsets of the result set of a seed query and identifies bugs when the DBMS's output violates the expected approximation relationship. EET [27] applies expression-level transformations that preserve query semantics, ensuring that transformed queries produce the same results as the original queries. Radar [45] detects metadata-related logic bugs in DBMSs by comparing query results between a database with metadata and a raw database without metadata but containing the same data, with any result inconsistency indicating a bug.

These works primarily focus on constructing equivalent query logic or applying different constraints to equivalent data. In contrast, EDC explores an alternative perspective by utilizing equivalence relationships between data to detect logic bugs, offering a complementary direction to existing approaches.

DBMS Test Case Generation. Many works have focused on generating diverse and effective test cases for DBMSs [20–22, 26, 29, 31, 41, 51, 53]. SQLsmith [41] incorporates the abstract syntax tree (AST) rules of SQL. SQUIRREL [53] introduces a novel intermediate representation to model SQL queries, allowing it to infer dependencies between queries. SQLRight [31] incorporates code coverage feedback, increasing the likelihood of uncovering logic bugs in rarely executed sections of DBMS code. Unicorn [51] generates test cases using hybrid input synthesis, which ensures both grammar correctness and the coverage of time-series features in queries to effectively test time-series databases. Griffin [22] uses a grammar-free mutation approach to test DBMSs by capturing the DBMS state information in a metadata graph. It mutates SQL queries based on this graph, ensuring that the mutations avoid semantic errors. LEGO [29] generates test cases by proactively exploring and analyzing type affinities between different SQL statement types, then synthesizing new SQL sequences based on discovered affinities to improve DBMS fuzzing coverage. DynSQL [26] utilizes dynamic query interaction to capture real-time DBMS state information, allowing for the incremental generation of complex and valid queries.

EDC complements these methods by focusing on detecting type-related issues through equivalent data construction. Existing techniques generate high-quality queries that can serve as valuable inputs for EDC to perform deeper analysis at the data level. In turn, EDC 's ability to uncover subtle data-centric bugs can enhance the overall effectiveness of these methods, suggesting strong potential for integration and mutual reinforcement between the approaches.

DBMS Test Data Generation. DBMS test data generation has also been a focus of some studies [16, 18, 19, 28, 32, 40, 42, 47, 48, 52]. ADUSA [28] is a query-aware database generator that produces both input data and the expected results for a given query, addressing the correctness oracle challenge in DBMS testing. It translates the schema and query into an Alloy specification, which is then solved to generate the necessary data. QAGen [32] generates a database that satisfies a given query and set of constraints by integrating traditional query processing techniques with symbolic execution. XData [19, 42] generates test data by using a constraint solver to create small, intuitive datasets that satisfy constraints identified from query mutations while ensuring coverage of all possible join trees with a number of datasets linear in the query size. EvoSQL [18] uses genetic algorithms with a fitness function based on the database engine's physical query plan to

ensure high coverage of query targets like `JOIN` and `WHERE`. DOMINO [16] generates test data for database schema testing using domain-specific operators, improving speed and fault detection over generalized search-based methods. TouchStone [48] enables parallel, memory-efficient query-aware generation, while Hydra [40] supports dynamic regeneration based on query workload summaries. SAM [52] leverages deep generative models to create high-fidelity databases that satisfy cardinality constraints, and Mirage [47] introduces graph distillation techniques using computation trees for efficient GNN training data generation.

While these existing techniques focus on generating data that satisfies query constraints or adheres to schema specifications, EDC takes a different approach by targeting logic bugs through the construction of equivalent data relationships across various data types. Furthermore, the high-quality data generated by these techniques could be integrated with EDC, further strengthening its ability to detect a broader range of bugs. By combining these approaches, testing can become more efficient and comprehensive, potentially contributing to improved robustness and reliability in DBMSs.

8 Conclusion

This paper introduces EDC, a novel framework for detecting logic bugs in DBMSs by leveraging data equivalence relationships. Unlike existing approaches that primarily focus on equivalent query structure or data constraints, EDC replaces data operation expressions with precomputed results, enabling precise identification of inconsistencies in query outputs. This expression-level substitution allows EDC to test whether semantically equivalent queries yield consistent outputs under actual DBMS execution, thereby exposing subtle inconsistencies that may indicate logic bugs. We evaluate EDC across seven widely used and well-tested DBMSs, applying it to a comprehensive set of type-operation combinations. The framework uncovered a significant number of logic bugs, many of which were previously unknown. Notably, a substantial portion of these bugs have been acknowledged by DBMS developers as critical issues, highlighting the practical value of expression-level equivalence testing in ensuring database correctness.

Acknowledgement

We appreciate the valuable comments provided by the re- viewers. This research is partly sponsored by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62525207, 62302256, 92167101, 62021002, U2441238), and CCF-ApsaraDB Research Fund(No. 2024007).

References

[1] 2024. ClickHouse Operators. https://clickhouse.com/docs/en/sql-reference/operators. Accessed: December 31, 2025.
[2] 2024. MDEV-34123. https://jira.mariadb.org/browse/MDEV-34123. Accessed: December 31, 2025.
[3] 2024. MySQL Functions and Operators. https://dev.mysql.com/doc/refman/8.0/en/built-in-function-reference.html. Accessed: December 31, 2025.
[4] 2024. Percona. https://github.com/percona/percona-server. Accessed: December 31, 2025.
[5] 2025. ClickHouse. https://clickhouse.com/. Accessed: December 31, 2025.
[6] 2025. EDC. https://github.com/THU-WingTecher/EDC. Accessed: December 31, 2025.
[7] 2025. Integer Types (Exact Value). https://dev.mysql.com/doc/refman/8.0/en/integer-types.html. Accessed: December 31, 2025.
[8] 2025. MariaDB. https://mariadb.org/. Accessed: December 31, 2025.
[9] 2025. MariaDB Data Types. https://mariadb.com/kb/en/data-types/. Accessed: December 31, 2025.
[10] 2025. MySQL. https://www.mysql.com/. Accessed: December 31, 2025.
[11] 2025. MySQL Data Types. https://dev.mysql.com/doc/refman/8.4/en/data-types.html. Accessed: December 31, 2025.
[12] 2025. OceanBase. https://github.com/oceanbase/oceanbase. Accessed: December 31, 2025.
[13] 2025. PostgreSQL. https://www.postgresql.org/. Accessed: December 31, 2025.

[14] 2025. SQL Data Types. https://www.databasestar.com/sql-data-types/. Accessed: December 31, 2025.

[15] 2025. TiDB. https://github.com/pingcap/tidb. Accessed: December 31, 2025.

[16] Abdullah Alsharif, Gregory M Kapfhammer, and Phil McMinn. 2018. DOMINO: Fast and Effective Test Data Generation for Relational Database Schemas. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 12–22.

[17] Daniel Bartholomew. 2014. *MariaDB cookbook*. Packt Publishing Ltd.

[18] Jeroen Castelein, Maurício Aniche, Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. 2018. Search-Based Test Data Generation for SQL Queries. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1220–1230.

[19] Bikash Chandra, Bhupesh Chawda, Biplab Kar, K. V. Maheshwara Reddy, Shetal Shah, and S. Sudarshan. 2015. Data Generation for Testing and Grading SQL Queries. 24, 6 (2015).

[20] Wenqian Deng, Jie Liang, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, and Yu Jiang. 2024. CONI: Detecting Database Connector Bugs via State-Aware Test Case Generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 26–37.

[21] 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*. 1–12.

[22] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-Free DBMS Fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[23] 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)*. 345–358.

[24] 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.

[25] Solid IT. 2022. DB-Engines Ranking of Relational DBMS. https://db-engines.com/en/ranking/relational+dbms. Accessed: December 31, 2025.

[26] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In *32nd USENIX Security Symposium*. 4949–4965.

[27] Zu-Ming Jiang and Zhendong Su. 2024. Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[28] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O Laleye, and Sarfraz Khurshid. 2008. Query-aware test generation using a relational constraint solver. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 238–247.

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

[30] Jie Liang, Zhiyong Wu, Jingzhou Fu, Yiyuan Bai, Qiang Zhang, and Yu Jiang. 2024. WingFuzz: Implementing Continuous Fuzzing for DBMSs. In *2024 USENIX Annual Technical Conference*. 479–492.

[31] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting logical bugs of DBMS with coverage-based guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. 4309–4326.

[32] Eric Lo, Carsten Binnig, Donald Kossmann, M Tamer Özsu, and Wing-Kai Hon. 2010. A framework for testing DBMS features. *The VLDB Journal* 19 (2010), 203–230.

[33] Roger C Lyndon. 1950. The representation of relational algebras. *Annals of mathematics* 51, 3 (1950), 707–729.

[34] PingCAP. [n. d.]. TiDB. https://github.com/pingcap/tidb. Accessed: December 31, 2025.

[35] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database management systems*. McGraw-Hill, Inc.

[36] 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*. 1140–1152.

[37] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[38] 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)*. 667–682.

[39] 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)*. 667–682.

[40] Anupam Sanghi, Raghav Sood, Dharmendra Singh, Jayant R Haritsa, and Srikanta Tirthapura. 2018. HYDRA: a dynamic big data regenerator. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1974–1977.

[41] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. 2018. SQLsmith: A Random SQL Query Generator. https://github.com/anse1/sqlsmith

[42] Shetal Shah, S. Sudarshan, Suhas Kajbaje, Sandeep Patidar, Bhanu Pratap Gupta, and Devang Vira. 2011. Generating Test Data for Killing SQL Mutants: A Constraint-Based Approach. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 1175–1186.

[43] Donald R Slutz. 1998. Massive Stochastic Testing of SQL. 98 (1998), 618–622.

[44] 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 *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2072–2084.

[45] Jiansen Song, Wensheng Dou, Yu Gao, Ziyu Cui, Yingying Zheng, Dong Wang, Wei Wang, Jun Wei, and Tao Huang. 2024. Detecting Metadata-Related Logic Bugs in Database Systems via Raw Database Construction. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1884–1897.

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

[47] Qingshuai Wang, Hao Li, Zirui Hu, Rong Zhang, Chengcheng Yang, Peng Cai, Xuan Zhou, and Aoying Zhou. 2024. Mirage: Generating Enormous Databases for Complex Workloads. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3989–4001.

[48] Qingshuai Wang, Yuming Li, Rong Zhang, Ke Shu, Zhenjie Zhang, and Aoying Zhou. 2022. A scalable query-aware enormous database generator for database evaluation. *IEEE Transactions on Knowledge and Data Engineering* 35, 5 (2022), 4395–4410.

[49] Michael Widenius, David Axmark, and Kaj Arno. 2002. *MySQL reference manual: documentation from the source*. " O'Reilly Media, Inc.".

[50] Zhiyong Wu, Jie Liang, Jingzhou Fu, Mingzhe Wang, and Yu Jiang. 2024. PUPPY: Finding Performance Degradation Bugs in DBMSs via Limited-Optimization Plan Construction. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 560–571.

[51] Zhiyong Wu, Jie Liang, Mingzhe Wang, Chijin Zhou, and Yu Jiang. 2022. Unicorn: Detect Runtime Errors in Time-series Databases with Hybrid Input Synthesis. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*. ACM, 251–262.

[52] Jingyi Yang, Peizhi Wu, Gao Cong, Tieying Zhang, and Xiao He. 2022. SAM: Database generation from query workloads with supervised autoregressive models. In *Proceedings of the 2022 International Conference on Management of Data*. 1542–1555.

[53] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing Database Management Systems with Language Validity and Coverage Feedback. In *The ACM Conference on Computer and Communications Security (CCS), 2020*.

Received April 2025; revised July 2025; accepted August 2025