SRS: Detecting Logic Bugs of Join Implementation in DBMSs via Set Relation Synthesis

JINHUI LAI, Nanchang University, China
CHI ZHANG, Tsinghua University, China
BINYAN LI, Nanchang University, China
CHENGLING LIANG, Nanchang University, China
JIE LIANG, Beihang University, China
ZHIYONG WU, Tsinghua University, China
JINGZHOU FU, Tsinghua University, China
YU JIANG, Tsinghua University, China
ZICHEN XU*, Nanchang University, China

Logic bugs can cause DBMSs to silently produce incorrect results for a given query, posing significant threats to software reliability and remaining challenging to detect. Join is a fundamental operation in DBMSs, enabling the combination of data from multiple tables; however, due to its complexity, it is also susceptible to logic bugs. Existing works detect logic bugs in join optimizations by altering query hints and system variables to alter the optimizer's choice of execution plans. However, these approaches struggle to detect logic bugs when query hints or system variables fail to influence the optimizer's behavior, or when the logic bugs reside in join implementation code that is unrelated to optimization. In this paper, we present **S**et **R**elation **S**ynthesis (SRS), a black-box testing approach that detects logic bugs of join implementation in DBMSs by leveraging set relations among different join operations. SRS applies transformations to the original join queries, including modifications to join types, join orders, and join conditions, while ensuring that the outputs of both the original and transformed queries preserve the expected set relations. Violations of these set relations indicate potential logic bugs. We realized SRS and evaluated it on five widely-used and extensively-tested DBMSs: MySQL, MariaDB, TiDB, PostgreSQL, and DuckDB. SRS uncovered 36 previously unknown and unique bugs, all of which have been confirmed, with 12 already fixed. Among these, 33 are logic bugs, demonstrating SRS's effectiveness and practicality in detecting logic bugs in the implementation of join operations within DBMSs.

CCS Concepts: • **Information systems → Database query processing**.

Additional Key Words and Phrases: DBMS Testing, Logic Bugs, Join Operation

ACM Reference Format:
Jinhui Lai, Chi Zhang, Binyan Li, Chengling Liang, Jie Liang, Zhiyong Wu, Jingzhou Fu, Yu Jiang, and Zichen Xu. 2025. SRS: Detecting Logic Bugs of Join Implementation in DBMSs via Set Relation Synthesis. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 363 (December 2025), 24 pages. https://doi.org/10.1145/3769828

*Zichen Xu is the corresponding author.

Authors' Contact Information: Jinhui Lai, jinhuilai@email.ncu.edu.cn, Nanchang University, China; Chi Zhang, chi-zhang@mail.tsinghua.edu.cn, Tsinghua University, China; Binyan Li, binyanli@email.ncu.edu.cn, Nanchang University, China; Chengling Liang, chenglinliang@email.ncu.edu.cn, Nanchang University, China; Jie Liang, liangjie.mailbox.cn@gmail.com, Beihang University, China; Zhiyong Wu, 253540651@qq.com, Tsinghua University, China; Jingzhou Fu, fuboat@outlook.com, Tsinghua University, China; Yu Jiang, jiangyu198964@126.com, Tsinghua University, China; Zichen Xu, xuz@ncu.edu.cn, Nanchang University, China.

ACM 2836-6573/2025/12-ART363
https://doi.org/10.1145/3769828

1 Introduction

Database management systems (DBMSs) are foundational software components in many critical applications, making their correctness of paramount importance. Join operations are the cornerstone of query processing in DBMSs, enabling the combination of data across multiple tables [7]. Specifically, every view definition, foreign-key enforcement, multi-table analytics, and recursive common-table expression ultimately decomposes into one or more join operations [20]. Implementing join operations is intrinsically complex, involving a range of sophisticated data-processing tasks (e.g., arithmetic and string evaluation, implicit type coercion, merge and nested-loop algorithms, and tight integration) with many other fundamental functionalities (e.g., `WHERE`, `AGGREGATION`) of the DBMS. In addition, the join operations contain several variants (e.g., inner join, outer join, semi join, anti join), each of which has its own semantic contract, further inflating the complexity of the DBMS's implementation of join. Therefore, ensuring the correctness of join implementations is paramount to the reliability of DBMSs.

However, logic bugs in the implementation of join operations can cause DBMSs to silently produce incorrect results, such as missing or duplicating rows. These logic bugs arise from the errors in join mechanisms (e.g., handling, duplicate elimination, hash partitioning, or implicit type coercion) due to their complexity. Figure 1 presents a test case that triggered a join-related logic bug in DuckDB, where the full outer join, denoted as q_1, returns the unexpected results. This bug is caused by an error in the join implementation when handling boundary values. In this test case, two tables, `t0` and `t1`, are created, each containing a column of type `VARCHAR`. Two tables are populated separately with two numeric boundary values: `-9876543210` and `-9876543210.0`. q_1 is then executed to concatenate `t0.c0` and `t1.c1` using a full outer join with the join condition `t0.c0 < t1.c1`. Based on the standard string comparison method, if one string is a prefix of the other, the shorter one is considered smaller than the longer one. Therefore, `-9876543210` is considered smaller than `-9876543210.0`. We expect this full outer join to produce one row, as the condition `t0.c0 < t1.c1` should evaluate to true. However, DuckDB returns an incorrect result for this query, where `t0.c0 < t1.c1` is incorrectly evaluated as false on the boundary values. Then two rows of incorrect results from the respective tables are retained independently in the resulting joined output, as shown in Figure 1.

TQS [28] and DQP [3] are two state-of-the-art testing approaches that target logic bugs in the optimization of join operations. Both approaches use the equivalence among alternative query execution plans as a test oracle to indicate the presence of logic bugs. By leveraging query hints and system variables, they manipulate the behavior of the query optimizer to ensure that the same query is executed using different plans. Any inconsistency in the query results may indicate the presence of logic bugs in join optimization. However, this strategy becomes less effective for detecting logic bugs in join operations when query hints or system variables fail to influence the optimizer's behavior, or when the logic bugs reside in parts of the join implementation unrelated to optimization. To detect the logic bug shown in Figure 1, we applied all query plan transformation techniques from both TQS and DQP to the bug-inducing query, i.e., q_1; however, no inconsistencies were observed. Therefore, neither approach was able to identify this bug.

To tackle logic bugs in the implementation of join in DBMSs, we propose a testing approach named **S**et **R**elation **S**ynthesis (SRS). The key insight is that different join operations involve various join types, orders, and constraints, which are not mutually independent but instead follow specific set relations. SRS thoroughly tests join operations by transforming join operations within queries—enabling the execution of various code paths related to join processing in the DBMS—and then detects logic bugs by checking whether the resulting join sets satisfy expected set relations. We

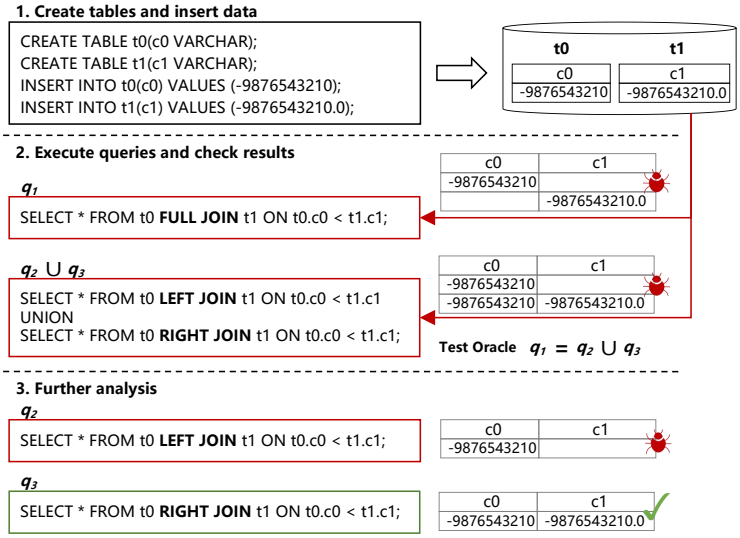

Fig. 1. An example that triggers a logic bug in DuckDB, where the full join query returns incorrect results.

develop a set relation model encompassing various join operations, such that any violation of the model signifies a logic bug.

For example, to detect the logic bug shown in Figure 1, we apply a join type transformation to query q_1, replacing its full outer join with a left outer join and a right outer join. We then construct the union query $q_2 \cup q_3$ based on the set relation that the union of a left outer join and a right outer join—using the same join condition—should be equivalent to a full outer join. The test oracle is the set relation $q_1 = q_2 \cup q_3$. The violation of this set relation enables us to identify this bug. Further analysis revealed that the full outer join query q_1 and the left outer join query q_2 produced incorrect results, while the right outer join query q_3 yielded correct results, thereby violating the set relation $q_1 = q_2 \cup q_3$. We reported this issue to the developers and received confirmation.

We implemented SRS as a DBMS testing tool and tested five mature and widely-used DBMSs with it, including MySQL [21], MariaDB [18], TiDB [29], PostgreSQL [22], and DuckDB [8]. All of these DBMSs have been extensively tested by numerous state-of-the-art testing approaches, so any newly discovered logic bug is likely one that has been missed by existing approaches. In total, we found 36 previously unknown and unique bugs: 15 in MySQL, 8 in MariaDB, 8 in TiDB, 2 in PostgreSQL, and 3 in DuckDB. All of these bugs have been confirmed by developers, with 12 already fixed and 33 identified as logic bugs. These results demonstrate the effectiveness of SRS in uncovering logic bugs in DBMSs.

Overall, we make the following contributions:

- We propose an idea for detecting logic bugs in the implementation of join operations within DBMSs by leveraging set relations among different join operations.
- We realize this idea in a black-box approach SRS, which detects logic bugs in the join implementation of DBMSs by transforming join queries based on our set relation model among different joins. The violation of the predefined set relations between the results of the original and transformed queries indicates the presence of a logic bug.
- We implemented and evaluated the SRS on five widely used DBMSs, uncovering 36 previously unknown unique bugs. Furthermore, we conducted comprehensive comparisons with the state-of-the-art DBMS testing approaches.

2 Preliminaries

2.1 Join Operations

Join operations are fundamental and essential in DBMSs [7, 20], facilitating the combination of data from multiple tables. To support complex data combination requirements, join operations include multiple types and may involve intricate conditions. In addition, join operations across multiple tables can produce various join orders, which may not only differ in performance but also yield inconsistent results, leading to logic bugs. These different joins are not mutually independent; rather, their results are expected to conform to specific set relations. To better illustrate the set relations constructed through transforming join operations, we first describe each type of join operation using relational algebra from a set-theoretic perspective in this section. Since the syntax of join operations may vary slightly across different DBMSs, we follow the classification of Mishra and Eich [20] in our description. It is worth noting that the term "join" in this paper refers to the SQL-level join operation, not to specific join mechanisms such as Hash Join, Nested Loops Join, or Sort-Merge Join.

We begin by introducing key relational algebra concepts: A **table (relation)** is a set of rows following a defined schema. Let R and T be tables with schemas $r(a_1, \ldots, a_n)$ and $t(b_1, \ldots, b_m)$, where r and t denote their rows (tuples), and a, b are columns (attributes). We then detail each join operation as follows:

The **cross join** corresponds to the Cartesian product of two tables, combining each row from the first table with every row from the second. The result set of the Cartesian product $R \times T$ can be denoted as follows:

$$R \times T = \{rt \mid r \in R \wedge t \in T\} \tag{1}$$

The **theta join (inner join)** extends the cross join by adding a join condition. The inner join of tables R and T can be denoted as $R \bowtie_c T$, where c is a **join condition**, which can be defined as $r(a)\theta t(b)$, $r(a)\theta e$, $t(b)\theta e$, or a combination of these cases. Here, θ is a comparison operator from $\{<, <=, =, !=, >=, >, \ldots\}$, $r(a)$ is a column of R, $t(b)$ is a column of T, and e is a valid value. The result set of $R \bowtie_c T$ can be denoted as follows:

$$R \bowtie_c T = \{rt \mid r \in R \wedge t \in T \wedge c\} \tag{2}$$

The **semi join** selects rows from one table that relate to at least one row in another table, based on a specified condition. The left semi join between two tables R and T by condition c can be denoted as $R \ltimes_c T$, and the right semi join can be denoted as $R \rtimes_c T$. The result set of $R \ltimes_c T$ can be denoted as follows:

$$R \ltimes_c T = \{r \mid r \in R \wedge t \in T \wedge c\} \tag{3}$$

The **anti join** selects rows from one table that do not satisfy the join condition with any row in another table. The left anti join between tables R and T by a condition c can be denoted as $R \triangleright_c T$, and the right anti join can be denoted as $R \triangleleft_c T$. The result set of $R \triangleright_c T$ can be denoted as follows:

$$R \triangleright_c T = \{r \mid r \in R \wedge \forall t \in T, \neg c\} \tag{4}$$

The **outer joins** ensure that non-matching rows from one or both tables are included in the result set. There are three types: left, right, and full outer join, which retain unmatched rows from the first table, the second table, or both, respectively. Their corresponding symbols are $⟕$, $⟖$, and $⟗$. The result set of the full outer join $R ⟗_c T$ can be denoted as follows:

$$\begin{aligned} R ⟗_c T = {} & \{rt \mid r \in R \wedge t \in T \wedge c\} \\ & \cup \{pad(r, null) \mid r \in R \wedge \neg c\} \\ & \cup \{pad(null, t) \mid t \in T \wedge \neg c\} \end{aligned} \tag{5}$$

Where $pad(r, null)$ extends the row r with null values for all T's columns, and $pad(null, t)$ extends the row T with null values for all R's columns.

The **natural join** combines tables by matching shared columns, keeping one copy of shared columns. The natural join of R and T can be denoted as $R * T$ and its result set is given by:

$$R * T = \{rt - t(a) \mid r \in R \wedge t \in T \wedge r(a) = t(a)\} \tag{6}$$

The join operations discussed above are categorized based on how two tables are connected. However, when a table is joined with itself (e.g., R cross joins itself, $R \times R$), a special type of join called a **self-join** is introduced.

Computing set relations among different join operations. The result of a join operation can be viewed as a temporary table constructed from the current state of the database, where each row in the temporary table is formed by concatenating rows from the original tables. For instance, if one join operation results in the concatenation of row r from table R and row t from table T, and another join operation yields the same concatenation, we consider there to be an intersection between these two join operations.

2.2 Set Relations Under SELECT Queries

The `SELECT` statement forms the core of SQL's Data Query Language (DQL), enabling data retrieval from a database. It uses the `WHERE` and `HAVING` clauses to filter records, while `ORDER BY` and `LIMIT` control the sorting and quantity of the results. For data analysis, the `GROUP BY` clause partitions data into groups, and aggregate functions calculate summary statistics within them.

Execution order of the query language features. The language features in a `SELECT` query follow a specific order of execution. Common table expressions and derived tables are processed first, followed by join operations. Subsequently, the query executes the `WHERE`, `GROUP BY`, `HAVING`, `SELECT`, `ORDER BY`, and `LIMIT` clauses in sequence. Therefore, the set relations constructed through join operations can serve as inputs to other language features of a `SELECT` query; however, these set relations are not always preserved after computations involving such language features.

Set relations under the query language features. In this paragraph, we discuss the set relations that query language features should adhere to. The computation of set relations is likewise based on the rows of the tables, rather than the specific values within those rows. Suppose there exists a set relation $\mathbb{R}_I$ between two inputs I_1 and I_2 to the given language feature. We investigate whether there exists a corresponding relation $\mathbb{R}_O$ such that the outputs O_1 and O_2 produced by the language feature preserve this relation:

Given that the `WHERE` and `HAVING` clauses operate as filters on the input data, the resulting output must retain the same set relation as the input—formally, $\mathbb{R}_I = \mathbb{R}_O$. The `ORDER BY` clause only modifies the order of the data without altering the data itself; therefore, the set relations between the input and output should remain equivalent. The `GROUP BY` clause automatically eliminates duplicates based on the grouping columns, so the output is expected to preserve only the equivalent set relation. The `LIMIT` clause restricts the number of records returned in the result. Therefore, when $\mathbb{R}_I$ represents a disjoint set relation, the output should preserve the same set relation. However, if $\mathbb{R}_I$ represents any other type of set relation, the output may not exhibit a deterministic set relation. Since the purpose of the `SELECT` clause is to output data that meets specified conditions in a format desired by the user, the resulting relational properties of the output are often difficult to determine. When it simply returns the original data without modification, the set relation remains unaffected. The only certainty is that if the input satisfies an equivalence relation, the output will also preserve that equivalence. For some aggregate functions, certain union relations may also be preserved, which we will discuss in more detail in Section 3.4.

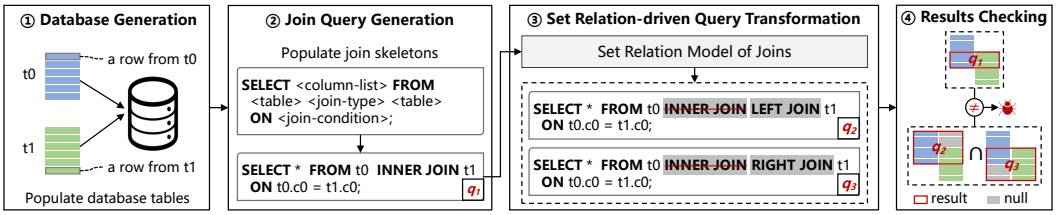

Fig. 2. Overview of SRS. The cells in the table represent rows, and the red-highlighted cells in step ④ denote the result set of each join operation. These results are expected to satisfy a set relation and can further serve as inputs to subsequent language features (e.g., `WHERE`, `GROUP BY`, `HAVING`) in the query.

2.3 Metamorphic Testing

Metamorphic testing [6] is a technique for constructing test oracles to detect logic bugs based on *metamorphic relations*. It generates follow-up test cases from existing ones and their results, and the outcomes of both the original and the follow-up tests are expected to conform to the specified metamorphic relations. A discrepancy indicates the presence of a logic bug. For example, in a sine computation program, the relation $\sin(x) = \sin(x + 2\pi)$ can be used to create follow-up tests that validate correctness.

Our work leverages this metamorphic testing technique to construct test oracles. Figure 2 provides an example: we begin with an original query involving an inner join operation and compute its result set S_{q_1}. We then generate follow-up queries by transforming the inner join to left/right outer joins, yielding results S_{q_2} and S_{q_3}, respectively. These results are expected to satisfy a set relation: $S_{q_1} = S_{q_2} \cap S_{q_3}$. Any violation of this relation indicates a potential logic bug.

3 Approach

In this paper, we propose **S**et **R**elation **S**ynthesis (SRS), a black-box approach for detecting logic bugs in the implementation of join operations within DBMSs, based on the set relations among different join operations. The key insight behind our approach is that different join operations involve various join types, orders, and constraints, which are not mutually independent but instead follow specific set relations. Specifically, SRS first generates an original query containing randomly constructed join operations, then derives one or two follow-up queries based on our set relation model. The results of these follow-up queries are expected to satisfy predefined set relations with the result of the original query. Any violation of these expected set relations signals a potential logic bug in the DBMS's join implementation.

3.1 Overview

Figure 2 illustrates the overview of SRS. In step ①, we randomly generate the database state with syntax-rule-based generators, which is a common practice in existing works [23–25]. In Step ②, the original query is constructed by randomly selecting the join type, the participating tables, and the join condition to populate join skeletons. In step ③, we transform the join operations in the original query based on a manually extracted set relation model of different join operations. This transformation results in one or two transformed queries, depending on the synthesis rules of the set relation model. In step ④, we check whether the results of the original query and the transformed query reflect the set relation defined by the set relation model of the corresponding join operations. Any discrepancy indicates that the test case has potentially triggered a logic bug.

We will provide a detailed explanation of the set relation model of joins we constructed, the transformations applied to join operations based on this model, and the criteria used to determine whether a logic bug has been triggered.

Table 1. Four categories set relations of joins and their synthesis rules.

ID	Set Relation	Synthesis Rule	Description	Derive From
R01	$S_{q_1} = S_{q_2}$	$R \bowtie_c T = \sigma_c(R \times T)$	Inner join equals cross join with a selection operation	Eq. 1, 2
R02	$S_{q_1} = S_{q_2}$	$R \,\psi_c\, T = T \,\psi_c\, R$	Reorder certain joins (ψ) does not affect the result	Eq. 1, 2, 5, 6
R03	$S_{q_1} = S_{q_2}$	$R \,\Psi_c\, T = R \,\Psi_{c'}\, T$	Equivalent c, c' in the same join yield the same result	Eq. 2- 5
R04	$S_{q_1} = S_{q_2}$	$R \ltimes_c T = \pi_{R.*}(R \bowtie_c T)$	Semi join equals inner join on one-sided	Eq. 2, 3
R05	$S_{q_1} = S_{q_2}$	$\pi_{C_1}(\ldots \pi_{C_n}(R \,\phi\, T)) = \pi_{C_1}(R \,\phi\, T)$	For any join ϕ, the final projection determines the output	Eq. 1 - 6
R06	$S_{q_1} = S_{q_2} \cap S_{q_3}$	$R \bowtie_c T = (R \rtimes_c T) \cap (R \ltimes_c T)$	Left and right outer joins intersect to an inner join	Eq. 2, 5
R07	$S_{q_1} = S_{q_2} \cap S_{q_3}$	$R \bowtie_{c_1} T = (R \bowtie_{c_2} T) \cap (R \bowtie_{c_3} T)$	Conditions $c_1 = c_2 \wedge c_3$ in inner join yield a intersection	Eq. 2
R08	$S_{q_1} \cap S_{q_2} = \varnothing$	$(R \ltimes_c T) \cap (R \triangleright_c T) = \varnothing$	Semi join and anti join yield disjoint result	Eq. 3, 4
R09	$S_{q_1} \cap S_{q_2} = \varnothing$	$(R \bowtie T) \cap (R \bowtie_{\neg c} T) = \varnothing$	Opposite c, $\neg c$ in inner join yields disjoint result set	Eq. 2
R10	$S_{q_1} = S_{q_2} \cup S_{q_3}$	$R \rtimes\ltimes_c T = (R \rtimes_c T) \cup (R \ltimes_c T)$	Full outer join is the union of left and right outer joins	Eq. 5
R11	$S_{q_1} = S_{q_2} \cup S_{q_3}$	$R = (R \ltimes_c T) \cup (R \triangleright_c T)$	Semi join and anti join yield complementary result	Eq. 3, 4
R12	$S_{q_1} = S_{q_2} \cup S_{q_3}$	$R \bowtie_{c_1} T = (R \bowtie_{c_2} T) \cup (R \bowtie_{c_3} T)$	Conditions $c_1 = c_2 \vee c_3$ in inner join yield a union set	Eq. 2

* $\psi \in$ { cross join, inner join, full outer join, natural join}, $\Psi \in$ { inner join, semi join, anti join, outer joins }, C_i is the columns of table

3.2 Set Relation Model of Joins

Set relations between different join operations form the foundation for constructing test oracles in the SRS approach. To effectively detect potential logic bugs in different join operations, we identify four categories of set relations—equivalence, intersection, disjointness, and union—and a total of 12 synthesis rules for constructing join queries that should conform to these relationships: 5 for equivalent relation, 2 for intersecting relation, 2 for disjoint relation, and 3 for union relation. Each rule is derived from the concepts introduced in Section 2 and summarized in Table 1.

3.2.1 Equivalent Relation, i.e., $S_{q_1} = S_{q_2}$. Join queries may yield identical results despite syntactic differences if they are semantically equivalent. We highlight three key scenarios: (1) Equivalence among join types: an explicit inner join is equivalent to a cross join followed by a selection (R01 in Table 1). (2) Equivalence among join orders: changing the join order of certain join operations preserves the result set (R02 in Table 1). (3) Equivalence among join conditions: applying logically equivalent join conditions (e.g., `t0 JOIN t1 ON TRUE` is equivalent to `t0 JOIN t1 ON NOT FALSE`) to the same join operation produces identical outputs (R03 in Table 1).

Additionally, some join queries may yield identical results for specific columns, even in the absence of full semantic equivalence. For example, an inner join followed by a one-sided projection—where the projection selects a subset of columns—is equivalent to a semi join (R04 in Table 1). More generally, when applying a sequence of projections to the same relation, only the final projection determines the output (R05 in Table 1).

3.2.2 Intersecting Relation, i.e., $S_{q_1} = S_{q_2} \cap S_{q_3}$. Intersection relations commonly arise in join operations, as certain join types and conditions naturally produce overlapping result sets. We highlight two key scenarios: (1) Intersecting join types: a classic example is the intersection of a left outer join and a right outer join (Figure 2), which can yield the inner join (R06 in Table 1). (2) Intersecting join conditions: join conditions such as <=, >=, and = often result in overlapping partitions, leading to intersecting outputs (R07 in Table 1).

3.2.3 Disjoint Relation, i.e., $S_{q_1} \cap S_{q_2} = \varnothing$. Specifically, a disjoint relation can also be regarded as a special case of an intersecting relation where the intersection is empty. We highlight two key scenarios: (1) Disjoint join types: joins with opposite semantics, such as semi join and anti join, produce mutually exclusive result sets, ensuring disjointness (R08 in Table 1). (2) Disjoint join conditions: in inner joins, logically contradictory conditions (i.e., c and $\neg c$, such as > vs. <=, or < vs. >=) can result in disjoint outputs (R09 in Table 1). However, this property does not generalize to other join types like outer joins, which preserve all rows from one table regardless of the condition.

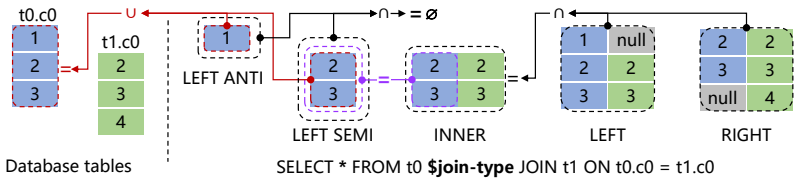

Fig. 3. An example of set relations among different join operations, given fixed tables and predicates. The union of a left anti join and a left semi join is equal to the left table of the join. The intersection of the left anti join and the left semi join is an empty set. The left semi join is the projection of the inner join on the columns of the left table. Intersecting a left join and a right join yields the inner join.

3.2.4 Union Relation, i.e., $S_{q_1} = S_{q_2} \cup S_{q_3}$. Union relations are also common in joins, as certain join types and conditions naturally produce complementary result sets. We highlight two key scenarios: (1) Union of join types: a full outer join is equivalent to the union of a left outer join and a right outer join over the same relations (R10 in Table 1). Semi join and anti join are complementary join operations; their union reconstructs the original relation (R11 in Table 1). Since one side of an outer join equals the original relation and an inner join followed by a one-sided projection is equivalent to a semi join, this rule can be further generalized. (2) Union of join conditions: join conditions such as <, =, and <= can form union relations under specific logical constraints (R12 in Table 1).

To better illustrate the four categories of set relations discussed above, we refer to the example in Figure 3. As shown, tables `t0` and `t1` are joined based on matching column values, with each arrow color representing a distinct set relation. The union of the left anti join and the left semi join is equivalent to the left input table, while their intersection is an empty set. The left semi join can be interpreted as the projection of the inner join onto the columns of the left table. The intersection of a left outer join and a right outer join yields the result of an inner join.

Generalization to multi-table joins. Multi-table and nested joins extend standard join operations by supporting additional join types and enabling the combination of multiple tables. Since the result of a two-table join can be regarded as an intermediate table, a multi-table join can be interpreted as a sequence of two-table joins, where each step joins the intermediate result with another base table. Therefore, by applying the synthesis rules in Table 1 to each two-table join within a multi-table join, we can construct the corresponding set relations systematically.

3.3 Set Relation-driven Join Transformation

To construct the set relations shown in Table 1, we propose four set relation-driven join transformations. Figure 4 shows examples of four set relation-driven join transformations, including modifying projection, modifying join order, modifying join type, and modifying join condition. We detail each as follows:

Modifying projection. This group of transformations adjusts the columns included in the result set. In some cases, join queries may produce identical outputs for specific columns, even if they are not fully semantically equivalent. For example, an inner join followed by a one-sided projection is equivalent to a semi join. More generally, projection operations can be composed without altering the number of rows or the values of columns that are retained across projections.

Modifying Join Order. This group of transformations leverages the principle that reordering join operations preserves query semantics. Join order modifications can be achieved through several practical approaches: (1) Join types enforcement. Some join types like `STRAIGHT_JOIN` can enforce an exact join order (e.g., `t0 STRAIGHT_JOIN t1` executes strictly left-to-right). (2) Query hints. DBMS like MySQL and MariaDB support hints (e.g., `/*+ JOIN_ORDER()*/`) to suggest a specific join order. (3) Manual reordering. Swapping table positions in SQL syntax may change join order. For example,

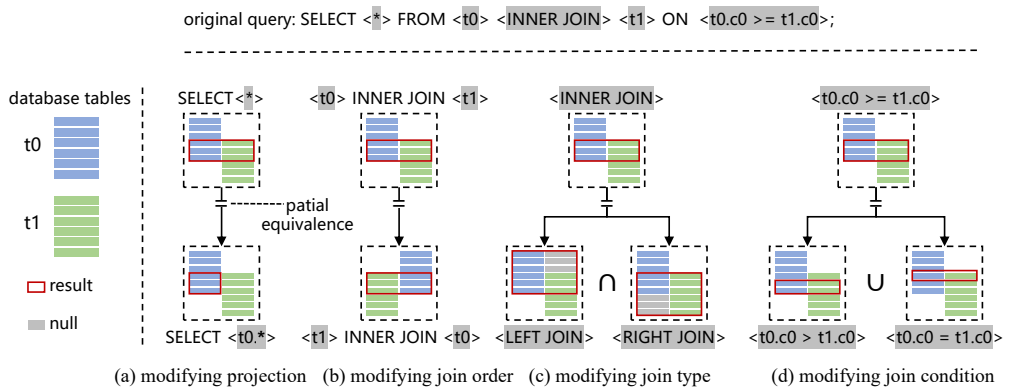

Fig. 4. **Examples of four set relation-driven join transformations, including: modifying projection, modifying join order, modifying join type, and modifying join condition**.

`t0 INNER JOIN t1` becomes `t1 INNER JOIN t0`. (4) Symmetric join conversion. Left/right outer joins can be logically inverted (e.g., `t1 LEFT JOIN t2` ≡ `t2 RIGHT JOIN t1`).

Modifying join type. This group of transformations modifies the join types to change the resulting join set. These rules can be classified into four categories based on their impact on the join set: (1) Preserving the join set—for instance, converting an explicit inner join into an implicit one without affecting the output. (2) Expanding the join set—such as transforming an inner join into a combination of left and right outer joins, where the original result can be recovered through intersection. (3) Inverting the join set—certain joins, like semi join and anti join, are complementary; a semi join can be rewritten as an anti join, resulting in disjoint outputs. (4) Reducing the join set—for example, decomposing a full outer join into left and right outer joins, which can be combined to reconstruct the original result.

Modifying join condition. This group of transformations modifies the join conditions and can also be classified into four categories, like modifying join type rules we presented above: (1) Preserving the join set—for example, replacing a condition like `JOIN ON TRUE` with `JOIN ON NOT FALSE`, which leaves the semantics unchanged. (2) Expanding the join set—for example, relaxing a condition from = to >= or <= can include additional rows. (3) Inverting the join set—for example, switching a condition from `JOIN ON t0.c0 IS NULL` to `JOIN ON t0.c0 IS NOT NULL`, which reverses the comparison logic and produces a complementary result. (4) Reducing the join set—for example, changing a condition from >= or <= to =, thereby narrowing the match criteria. These transformations allow flexible control over query behavior while preserving logical consistency. Notably, modifying join conditions is primarily suitable for inner joins, as other types of joins—such as outer joins—may produce overlapping result sets, making the effects of such modifications harder to predict. Additionally, the influence of `NULL` values must be carefully considered, as addressed in TLP[24].

Algorithmic sketch of applying transformation. Algorithm 1 shows the procedure for applying transformations to an original query in order to generate the corresponding follow-up transformed queries. For each generated original query, we first determine the join type *join_type* used in the query using the `GetJoinType` function. To account for the fact that not all language features support every type of set relation, we extract the language features $features$ present in the original query using the `GetFeaturesInQuery` function. A detailed explanation is provided in Section 3.4. Next, we retrieve the corresponding set relation synthesis rules *ruleList* related to *joinType* with function `SetRelationModel`, which are shown in Table 1. For each join type, there

Algorithm 1: Set relation-driven join transformation

Input: original query q_1
Output: transformed queries q_2 or q_2, q_3

```
joinType ← GetJoinType(q1);
features ← GetFeaturesInQuery(q1);
ruleList ← SetRelationModel(joinType);
singleRule ← Rand(rulesOfJoin);
if not CheckSupport(singleRule, features) then
    return null;
end
trans ← GenTrans(singleRule);
q2 or q2, q3 ← DoTrans(q1, trans);
return q2 or q2, q3
```

may be multiple applicable rules; we use the `Rand` function to randomly select one of them and generate a single rule *singleRule*. When a query's language features do not support the set relation specified in the rule (as checked by the `CheckSupport` function), the query is discarded and a new one is generated. Then, based on the selected *singleRule*, we use the `GenTrans` function to generate a set of transformations, denoted as *trans*. Since different DBMSs support varying semantics of joins, there is no fixed mapping between rules and transformations. Moreover, a single rule may correspond to multiple transformations required to modify the join in the original query. Finally, we apply the transformations to the original query q_1 using the `DoTrans` function to generate the transformed queries. Depending on the specific rules in our set relation model, a transformed query may result in either one query or two queries.

3.4 Results Checking

The core of SRS is to transform join operations in a way that yields set relations consistent with the set relation model. These set relations are expected to affect the computation of downstream language features and manifest in the final query output. However, if the output of a language feature cannot reflect the set relation expressed by its input, that feature cannot be used to construct test cases for SRS. For example, a query with a `LIMIT` clause cannot be used as a test case when the test oracle is based on equivalence relations, as two equivalent joins may produce identical results with differing row order, leading to different retained values in the query results. We provide an analysis of the set relations supported by various language features, and further describe the scope of the language features that our approach supports.

`WHERE`. `WHERE` clause is used to filter data based on specified conditions. Given a selection condition p, applying the filtering operation σ_P to both sides of any equation in Table 1 preserves the set relations. Therefore, for any set relation input to the `WHERE` clause, the output is expected to preserve the same set relation.

`HAVING`. `HAVING` performs filtering in a manner similar to `WHERE`. Therefore, the set relation of the input to `HAVING` is likewise expected to be preserved in its output.

`GROUP BY`. Since `GROUP BY` removes duplicates based on the grouping columns, when the input adheres to an equivalent set relation, the output is expected to preserve the same set relation.

`SELECT`. The `SELECT` clause specifies the structure of the query output. When it simply returns the original data without modification, the set relation remains unaffected. However, when more complex computations or aggregate functions are applied to the query results, the analysis of set

relations becomes more intricate. When two join queries yield equivalent set relations (R01–05 in Table 1), applying the aggregate function or complex computations to each will produce identical outputs. When three join queries adhere union relation (i.e., $S_{q_1} = S_{q_2} \cup S_{q_3}$, R10–12 in Table 1), the behavior of aggregate functions is as follows:

$$\begin{cases} \pi_{\text{MAX}(col)}(S_{q_1}) = \max(\pi_{\text{MAX}(col)}(S_{q_2}), \pi_{\text{MAX}(col)}(S_{q_3})) \\ \pi_{\text{MIN}(col)}(S_{q_1}) = \min(\pi_{\text{MIN}(col)}(S_{q_2}), \pi_{\text{MIN}(col)}(S_{q_3})) \\ \pi_{\text{COUNT}(*)}(S_{q_1}) = \pi_{\text{COUNT}(*)}(S_{q_2}) + \pi_{\text{COUNT}(*)}(S_{q_3}) \\ \pi_{\text{SUM}(col)}(S_{q_1}) = \pi_{\text{SUM}(col)}(S_{q_2}) + \pi_{\text{SUM}(col)}(S_{q_3}) \\ \pi_{\text{AVG}(col)}(S_{q_1}) = \dfrac{\pi_{\text{SUM}(col)}(S_{q_2}) + \pi_{\text{SUM}(col)}(S_{q_3})}{\pi_{\text{COUNT}(*)}(S_{q_2}) + \pi_{\text{COUNT}(*)}(S_{q_3})} \end{cases}$$

Where col is a common column of S_{q_1}, S_{q_2}, and S_{q_3} whose data type supports these aggregate functions, and $*$ denotes all columns. Therefore, if a `SELECT` clause returns all columns directly, it should preserve all set relations. When using the aggregate functions listed above, the results align with those derived from union relations; otherwise, only equivalence relations are preserved.

`ORDER BY`. The `ORDER BY` clause is used to sort the output of a query. Since it does not add, remove, or modify data, it has no impact on the underlying set relation. Therefore, for any set relation input to the `ORDER BY` clause, the output is expected to preserve the same set relation.

`LIMIT`. The `LIMIT` clause controls the number of results returned by a query. However, since the order of query results may be affected by various factors, it is often uncertain which specific rows will be retained after applying `LIMIT`. Consequently, many set relations may not be preserved under this operation. Nonetheless, the disjoint relation remains applicable to the `LIMIT` clause: if two set relations are disjoint, applying the `LIMIT` clause to both will still yield disjoint results.

Summary. For the `WHERE`, `HAVING`, and `ORDER BY` clauses, their generation poses no limitation, as these clauses are expected to preserve the same set relations. For the `GROUP BY` clause, only equivalent relations can be supported. For the `LIMIT` clause, only disjoint relations can be supported. For the `SELECT` clause, there is no limitation if it simply returns the original data without modification. However, for more complex cases, only equivalent and union relations can be supported. Finally, we identify potential logic bugs by detecting violations of the expected set relations among the results of the original and transformed queries.

4 Implementation

We implemented SRS on top of SQLancer[1], a DBMS testing framework designed for the random generation of database states and SQL queries, which also supports multiple test oracles [23, 24]. In total, SRS comprises 5,500 lines of code. This code is primarily used to generate more complex join operations, upon which transformations are applied based on the synthesis rules. Despite this, the approach remains simple and can be easily adapted to other testing frameworks, such as SQLsmith [1]. In this section, we present the technical details of SRS's implementation.

Database state generation. Following the standard pipeline in popular DBMS testing tools like SQLancer, we generate the database state first to improve semantic correctness by generating state-conforming queries. SRS applies the automated, syntax-rule-based random generation approach of SQLancer to ensure that the database state exhibits sufficient diversity, thereby enabling thorough and effective bug detection. SRS also incorporates several strategies, including the generation of diverse schemas and data characteristics, such as boundary values. The details of these strategies are further discussed in Section 6.

[1]https://github.com/sqlancer/sqlancer

Table 2. SQL grammar of join skeleton.

Explicit Join	
explicit-join	::= SELECT <column-list> FROM <table> <join-clause>
column-list	::= [<column> [, <column> ...]]*
join-clause	::= [<join-type> <table> ON <join-condition>]*
join-condition	::= <term> [(AND|OR) <term>]*
term	::= <column> <comparison-operator> (<column>|<value>)
Implicit Join	
implicit-join	::= SELECT <column-list> FROM <table-list> <join-clause>
column-list	::= [<column> [, <column> ...]]*
table-list	::= [<table> [, <table> ...]]*
join-clause	::= WHERE <join-condition>
join-condition	::= <term> [(AND|OR)<term>]*
term	::= <column> <comparison-operator> (<column>|<value>
column/value/table	::= [a-zA-Z_][a-zA-Z0-9_]*
join-type	::= INNER JOIN|LEFT JOIN|RIGHT JOIN|...
comparison-operator	::= >|>=|<|<=|=|!=|LIKE|NOT LIKE|EXISTS|NOT EXISTS|...

"<>" indicates the placeholders to be filled.

Join query generation. To generate join queries for constructing equivalent sets, SRS populates join skeletons to produce diverse join queries. Table 2 presents the SQL grammar of explicit and implicit join skeletons, which defines the structure for generating SQL queries involving join operations. Here, an implicit join query refers to language features that are automatically optimized into join operations by DBMSs. For instance, in MySQL, `EXISTS (subquery)` predicates can be transformed into semi joins[2]. This grammar outlines the essential components required to construct a valid query, including the selection list, table list, join types, and join conditions. The gray-highlighted areas in the table represent placeholders that can be filled with specific query elements to form complete SQL statements. For example, the skeleton corresponding to the join query in Figure 1 is `SELECT <column-list> FROM <table> <join-type> <table> ON <join-condition>`, where "<>" indicates the placeholders to be filled.

The random generation of each join query consists of three steps. First, SRS randomly selects a predefined join query skeleton based on the database state. Given a database state, SRS traverses all table columns to identify those with the same or similar types that satisfy the join condition, maintaining a record of joinable column pairs. Second, SRS enumerates all possible combinations of join types and conditions based on column pairs. This allows SRS to generate specific types of joins, such as self-joins, which occur when the same table is used on both sides of a join operation, with distinct aliases assigned to each instance. For example, in `SELECT * FROM node n LEFT JOIN node p ON n.parent = p.id`, table `node` self-joins itself via `LEFT JOIN`. This also enables SRS to cover as many join types as possible, thereby testing all the set relations proposed in this paper. Third, additional query clauses are randomly generated based on the table information in join operations. Although not all language features can preserve or reflect the input set relations in their output, we continue to use a fully random, syntax-rule-based generation approach for the rest of the SQL query. This approach decouples query generation from the set relation model, making it easier to extend the model with additional rules in the future. For the generation of other language features, aside from considering the tables involved in joins, our approach remains consistent with SQLancer's query generation strategy for NoREC test oracle [23].

[2]https://dev.mysql.com/doc/refman/8.0/en/semijoins.html

Table 3. SRS found 36 previously unknown, unique bugs across five mature DBMSs, all of which have been confirmed, with 12 already fixed.

ID	DBMS	Status	Severity	Bug Type	Join Type	Set Relation	Rule
1	MySQL	Confirmed	Non-critical	Logic Bug	Inner Join	$S_{q_1} = S_{q_2}$	R02
2	MySQL	Confirmed	Serious	Logic Bug	Inner Join	$S_{q_1} = S_{q_2}$	R02
3	MySQL	Confirmed	Serious	Logic Bug	Inner Join	$S_{q_1} = S_{q_2}$	R02
4	MySQL	Confirmed	Serious	Logic Bug	Inner Join	$S_{q_1} = S_{q_2}$	R03
5	MySQL	Confirmed	Serious	Logic Bug	Outer Join	$S_{q_1} = S_{q_2}$	R03
6	MySQL	Confirmed	Serious	Logic Bug	Inner Join	$S_{q_1} = S_{q_2}$	R04
7	MySQL	Confirmed	Serious	Logic Bug	Outer Join	$S_{q_1} = S_{q_2} \cap S_{q_3}$	R06
8	MySQL	Confirmed	Serious	Logic Bug	Outer Join	$S_{q_1} = S_{q_2} \cap S_{q_3}$	R06
9	MySQL	Confirmed	Serious	Logic Bug	Anti Join	$S_{q_1} \cap S_{q_2} = \varnothing$	R08
10	MySQL	Confirmed	Serious	Logic Bug	Anti Join	$S_{q_1} \cap S_{q_2} = \varnothing$	R08
11	MySQL	Confirmed	Serious	Logic Bug	Semi Join	$S_{q_1} = S_{q_2} \cup S_{q_3}$	R11
12	MySQL	Confirmed	Serious	Logic Bug	Semi Join	$S_{q_1} = S_{q_2} \cup S_{q_3}$	R11
13	MySQL	Confirmed	Serious	Logic Bug	Inner Join	$S_{q_1} = S_{q_2} \cup S_{q_3}$	R11
14	MySQL	Confirmed	Serious	Logic Bug	Inner Join	$S_{q_1} = S_{q_2} \cup S_{q_3}$	R11
15	MySQL	Confirmed	Serious	Logic Bug	Inner Join	$S_{q_1} = S_{q_2} \cup S_{q_3}$	R12
16	MariaDB	Fixed	Major	Logic Bug	Inner Join	$S_{q_1} = S_{q_2}$	R02
17	MariaDB	Fixed	Major	Logic Bug	Inner Join	$S_{q_1} = S_{q_2}$	R02
18	MariaDB	Fixed	Major	Logic Bug	Inner Join	$S_{q_1} = S_{q_2}$	R04
19	MariaDB	Fixed	Major	Logic Bug	Outer Join	$S_{q_1} = S_{q_2}$	R05
20	MariaDB	Fixed	Major	Logic Bug	Anti Join	$S_{q_1} \cap S_{q_2} = \varnothing$	R08
21	MariaDB	Fixed	Major	Logic Bug	Inner Join	$S_{q_1} = S_{q_2} \cup S_{q_3}$	R11
22	MariaDB	Fixed	Major	Logic Bug	Inner Join	$S_{q_1} = S_{q_2} \cup S_{q_3}$	R11
23	MariaDB	Fixed	Major	Logic Bug	Inner Join	$S_{q_1} = S_{q_2} \cup S_{q_3}$	R11
24	TiDB	Confirmed	Major	Logic Bug	Cross Join	$S_{q_1} = S_{q_2}$	R02
25	TiDB	Confirmed	-	Logic Bug	Outer Join	$S_{q_1} = S_{q_2}$	R05
26	TiDB	Confirmed	Moderate	Logic Bug	Outer Join	$S_{q_1} = S_{q_2}$	R05
27	TiDB	Confirmed	-	Logic Bug	Anti Join	$S_{q_1} \cap S_{q_2} = \varnothing$	R08
28	TiDB	Confirmed	-	Logic Bug	Semi Join	$S_{q_1} = S_{q_2} \cup S_{q_3}$	R11
29	TiDB	Confirmed	Moderate	Logic Bug	Anti Join	$S_{q_1} = S_{q_2} \cup S_{q_3}$	R11
30	TiDB	Confirmed	Major	Logic Bug	Inner Join	$S_{q_1} = S_{q_2} \cup S_{q_3}$	R11
31	TiDB	Confirmed	Moderate	Assertion Failure	-	-	-
32	PostgreSQL	fixed	-	Logic Bug	Inner Join	$S_{q_1} = S_{q_2}$	R01
33	PostgreSQL	fixed	-	Logic Bug	Inner Join	$S_{q_1} = S_{q_2} \cup S_{q_3}$	R12
34	DuckDB	Fixed	-	Logic Bug	Outer Join	$S_{q_1} = S_{q_2} \cup S_{q_3}$	R10
35	DuckDB	Confirmed	-	Error	-	-	-
36	DuckDB	Fixed	-	Assertion Failure	Inner Join	-	-

5 Evaluation

We evaluate the effectiveness and efficiency of SRS by answering the following research questions:

- **Q1. New Bugs**: How many unique and new bugs in real-world DBMSs can SRS detect?
- **Q2. Bugs Detected via Each Set Relation**: How effective is the set relation model in detecting logic bugs?
- **Q3. Comparison with the State-of-the-Art**: Can SRS outperform existing state-of-the-art approaches for detecting logic bug in DBMSs?

Table 4. Tested DBMSs.

DBMS	Version	Ranking	Github Stars
MySQL	9.2.0	2	11.3k
MariaDB	11.7.2	14	6k
TiDB	8.5.1	73	38.3k
PostgreSQL	17.5.1	4	18.1k
DuckDB	1.2.0	44	28.4k

5.1 Testing Setup

Tested DBMSs. We consider five DBMSs in our evaluation, including MySQL, MariaDB, TiDB, PostgreSQL, and DuckDB. According to the DB-Engine's Ranking[3], and GitHub, these DBMSs are among the most popular and widely-used DBMSs. These DBMSs are also commonly used in the evaluation of previous works [3, 23–25]. Therefore, any bugs identified by SRS may have been overlooked by existing approaches. We tested the latest stable release versions of each DBMS—MySQL (9.2.0), MariaDB (11.7.2), TiDB (8.5.1), PostgreSQL (17.5.1), and DuckDB (1.2.0). Since release versions are generally considered stable, any bugs identified in these DBMSs serve as strong evidence of the effectiveness of our approach.

These DBMSs represent different categories. MySQL, MariaDB, and PostgreSQL are traditional, relational, and client-server DBMSs, which are widely used in industry. TiDB is a distributed relational DBMS designed to support large-scale deployments, offering high availability and scalability in distributed environments. DuckDB is a lightweight embedded DBMS.

Environment. Experiments were conducted on a server equipped with an AMD EPYC 7773X 64-Core Processor and 1 TB of RAM, running Ubuntu 22.04.

5.2 New Bugs

Table 3 summarizes the 36 previously unknown, unique bugs detected by SRS, including 15 in MySQL, 8 in MariaDB, 8 in TiDB, 2 in PostgreSQL, and 3 in DuckDB. All bugs have been confirmed by developers, with 12 already fixed and 33 identified as logic bugs. These 36 bugs cover all four set relations and five out of the six join types—cross, inner, outer, semi, and anti—excluding natural join. We checked the testing log and found that the generated test cases do include natural joins but do not trigger natural-join-related bugs. This result indicates that our test case generation achieves complete coverage of both join types and set relations.

Bug types and join types. Among the 36 detected bugs, 33 are logic bugs. Additionally, SRS identified 3 other bugs—2 assertion failures and 1 error—that can be exposed without relying on set relation synthesis rules. All of the 33 logic bugs are triggered by join operations, as their minimized test cases all include at least one `JOIN`. Specifically, 17 involve inner join, 7 involve outer join, 5 involve anti join, 3 involve semi join, and 1 involve cross join. This does not imply each generated query contains only a single join operation. Rather, we minimize each test case to isolate the specific join operation responsible for triggering the bug.

Among 3 other bugs, bug #31 is an assertion failure triggered by an `UPDATE` statement; bug #35 is an error triggered by an `UPDATE` statement; bug #36 is an assertion failure triggered by an `INNER JOIN` query. Although SRS targets join-related logic bugs, it also generates a wide range of language features supported by the tested DBMSs. This allows us to uncover other types of bugs.

Although all the logic bug-inducing test cases involve join operations, the root causes of these bugs are diverse. On one hand, we received feedback indicating that 16 of the 33 logic bugs we

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

reported are related to the query optimizer. SRS transforms the order, type, and conditions of join operations, which can cause the optimizer to adopt different optimization strategies, ultimately triggering optimizer-related bugs. On the other hand, in one bug report, the developer explicitly stated that the root cause was entirely unrelated to the join operation. In another case, our analysis of the bug-inducing query—including its involved language features and the results of the join operation—also indicated that the root cause was independent of the join. This suggests that SRS can also detect logic bugs in the implementation of other language features.

Bug status. All bugs have been confirmed by developers, with 12 already fixed and 33 identified as logic bugs. In our bug reports for MariaDB, PostgreSQL, and DuckDB, developers were highly responsive in addressing the issues, indicating that these bugs may pose significant risks. In contrast, MySQL and TiDB required more time to fix reported bugs. This aligns with observations from previous works [3, 24], which limited our reporting of additional potential bugs to avoid duplication.

Bug importance. Most of the bugs found by SRS are critical and have a high likelihood of impacting real users. In MySQL, MariaDB, and TiDB, 27 bugs are classified as Serious, Major, or Moderate, highlighting their significant impact on the systems and underscoring the necessity for high-priority fixes.

Listing 1. A bug found in MariaDB via equivalent relation. Applying a projection to a join result should not alter the size or the values of certain columns, which violates rule R05 in Table 1.

```
-- Modifying the projection leads to an incorrect left outer join result.
CREATE TABLE t0(c0 FLOAT);
CREATE TABLE t1(c1 FLOAT);
INSERT INTO t0 VALUES (NULL), (NULL);
INSERT INTO t1 VALUES (1.0);
CREATE UNIQUE INDEX i0 USING BTREE ON t0(c0);
SELECT t1.c1 FROM t1 LEFT OUTER JOIN t0 ON t0.c0 IS NULL;
-- {1} 🐞, expect {1, 1}
SELECT * FROM t1 LEFT OUTER JOIN t0 ON t0.c0 IS NULL;
-- {(c1:1, c0:0), (c1:1, c0:0)} ✔
```

Listing 2. A bug found in MySQL via intersecting relation. The intersection of the results from a left outer join and a right outer join should be equivalent to the result of an inner join, violating rule R06 in Table 1.

```
-- "Left join intersect right join" produces an incorrect result.
CREATE TABLE t0(c0 FLOAT PRIMARY KEY);
CREATE TABLE t1(c1 INT PRIMARY KEY );
INSERT IGNORE INTO t0(c0) VALUES(0);
INSERT INTO t1(c1) VALUES(1);
SELECT * FROM t1 INNER JOIN t0 ON t0.c0 IS NULL; -- {} ✔
SELECT * FROM t1 LEFT JOIN t0 ON t0.c0 IS NULL INTERSECT
SELECT * FROM t1 RIGHT JOIN t0 ON t0.c0 IS NULL; -- {(c1:1, c0:0)} 🐞, expect {}
```

5.3 Bugs Detected via Each Set Relation

All 33 logic bugs were uncovered by SRS through the four types of set relations derived from our synthesis rules. Specifically, 14 bugs were uncovered via equivalent relations, 2 via intersecting relations, 4 via disjoint relations, and 13 via union relations. Among the 12 synthesis rules, 9 successfully revealed bugs, while R01, R07, and R09 require further investigation. According to our bug list in Table 3, the three most effective rules are R11, R02, and R08, which found 10, 6, and 4 bugs, respectively. To demonstrate the effectiveness of our synthesis rules summarized in Table 1 for detecting logic bugs, we present the following case studies.

Bug found via equivalent relation. Listing 1 shows bug #19 we discovered in MariaDB via the equivalent relation. In this case, a left outer join between tables `t0` and `t1` is expected to return `{1,1}` for the `c1` column, as evidenced by `SELECT *`. However, when the query projects only `c1`, the result becomes `{1}` instead of `{1,1}`, due to unintended deduplication. This behavior violates Rule R05. Moreover, this bug was confirmed and fixed by the MariaDB developers within 3 hours. This highlights the significance of the bug we uncovered and the prompt attention it received from the MariaDB development team.

Bug found via intersecting relation. Listing 2 presents bug #8, which we discovered in MySQL via the intersecting relation approach. According to rule R06, the intersection of the results from a left outer join and a right outer join should be equivalent to the result of an inner join. However, in this case, the query result violates this fundamental principle, exposing a logic bug. Interestingly, when either `c0` or `c1` is not defined as a `PRIMARY KEY`, the intersection of the left and right outer join produces the correct result. This highlights the complexity of join operations and their interactions with many other language features.

Listing 3. A bug found in TiDB via disjoint relation. The intersection of the results from a semi join and an anti join yields a non-empty set, which is expected to be empty, thereby violating rule R08 in Table 1.

```
-- Anti join's result is contradictory with semi join's.
CREATE TABLE t0(c0 BOOL ZEROFILL AS (-1) VIRTUAL , c1 DECIMAL);
CREATE TABLE t2 (c2 BOOL);
INSERT IGNORE INTO t0(c1) VALUES (NULL);
ANALYZE TABLE t0;
CREATE INDEX i0 ON t0(c0 ASC, c1 DESC);
INSERT INTO t2(c2) VALUES (1);
SELECT t2.c2 FROM t2 WHERE NOT EXISTS (SELECT 1 FROM t0 WHERE t2.c2 < t0.c0);
-- Anti join's result: {1} 🐞, expect {}
SELECT t2.c2 FROM t2 WHERE EXISTS (SELECT 1 FROM t0 WHERE t2.c2 < t0.c0);
-- Semi join's result: {1} ✔
```

Listing 4. A bug found in TiDB via the union relation. Since semi-join and anti-join are complementary operations, the union of the results from a semi join and an anti join produces an empty result, whereas it is expected to reconstruct the original input relation, thus violating rule R11 in Table 1.

```
-- "Semi join union anti join" produces an incorrect result
CREATE TABLE t0(c0 INT ZEROFILL AS (-1) VIRTUAL, c1 FLOAT, PRIMARY KEY(c1));
CREATE TABLE t2 (c2 INT);
INSERT IGNORE INTO t0(c1) VALUES (0);
CREATE INDEX i0 ON t0(c0, c1);
INSERT INTO t2(c2) VALUES (0);
SELECT * FROM t2 WHERE NOT EXISTS (SELECT 1 FROM t0 WHERE c2 = c0) UNION
SELECT * FROM t2 WHERE EXISTS (SELECT 1 FROM t0 WHERE c2 = c0);
-- {}, Semi join union anti join returns empty set 🐞, expect {0}
SELECT t2.c2 FROM t2; -- {0} ✔
```

Bug found via disjoint relation. Listing 3 shows bug #27 we found in TiDB via disjoint relation. In MySQL-based DBMSs, `EXISTS(subquery)` is optimized into a semi join, while `NOT EXISTS(subquery)` is optimized into an anti join[4]. In this test case, the first query, corresponding to an anti join, and the second query, corresponding to a semi join, both return the same result: 1. According to rule R08, the intersection of the results from a semi join and an anti join should be an empty set. However, in this case, both joins return the same value, violating this set relation due to a bug in TiDB's anti join implementation.

[4]https://dev.mysql.com/doc/refman/8.4/en/semijoins-antijoins.html

Bug found via union relation. Listing 4 shows bug #28 discovered in TiDB through the union relation. In this case, the union of the results from a semi join and an anti join unexpectedly returns an empty set, which contradicts the result of `SELECT t2.c2 FROM t2;` that correctly yields `0`. According to rule R11, the union of a semi join and an anti join should reconstruct the original relation. However, due to the semi join returning an incorrect result, the union output violates rule R11, revealing this logic bug in TiDB.

5.4 Comparison With the State-of-the-Art

To illustrate the effectiveness and efficiency of our approach, we compared SRS with state-of-the-art DBMS testing approaches at two levels: tools and test oracles. At the tool level, we compared the effectiveness and efficiency of tools implemented by baselines over 24 hours. However, due to the implementation quality, a tool may not fully reflect the true bug-detection capability of its underlying approach. Therefore, we further compared the test oracles proposed by different approaches from a theoretical perspective. This evaluation focuses on the detection of logic bugs, and the results include only the logic bugs found, as SRS and the baseline methods are specifically designed to uncover logic bugs in join operations. Other types of bugs (e.g., runtime errors or assertion failures) can typically be detected by most existing testing tools.

Baseline. We selected six state-of-the-art approaches for comparison: PQS [25], TLP [24], NoREC [23], Pinolo [11], EET [13], and DQP [3]. Both DQP and TQS [28] are approaches designed to detect logic bugs in join operations by leveraging the equivalence among different query plans for a given query. These two approaches are the most closely aligned with the goal of SRS. We did not include TQS in our comparison because its source code is unavailable, and the authors informed us via email that they are currently unable to share it. DQP is a simpler alternative to TQS and has specifically studied TQS, concluding that it offers the same level of bug-finding effectiveness [13] as TQS. Therefore, we assume that comparing with DQP effectively represents a comparison with this class of approaches. PQS generates queries based on the database state such that the result is guaranteed to include specific rows from the database. It supports testing of join operations. EET leverages tautologies and contradictions to construct equivalent queries, enabling the testing of any syntactic feature involving expressions. As a result, it also supports the testing of join operations. Both TLP and Pionlo leverage set-based relationships to detect logic bugs, but their transformation targets are expressions within query clauses. Both tools provide support for generating join operations. NoREC generates equivalent queries by exploiting the equivalence between the `WHERE` and `SELECT` clauses. While NoREC is not designed to specifically test join operations, it has previously detected join-related bugs. As such, we include a comparison with NoREC in our evaluation.

5.4.1 Comparison to Other DBMS Logic Bug Testing Tools. We run each tool on the latest version of each supported DBMS for a duration of 24 hours. Since the baseline approaches do not support all the DBMSs that SRS does, our comparison is limited to the DBMSs commonly supported by both SRS and each baseline. To ensure fair comparisons, we allocated four threads for each DBMS test in our experiments, following common practice in previous work [11]. In this experiment, we aim to investigate the effectiveness and efficiency of SRS in comparison to state-of-the-art approaches.

Results of effectiveness. Table 5 presents the number of unique logic bugs identified by each approach on each DBMS in the duration of 24 hours. Since not all approaches target join-related logic bugs, we did not exclude non-join-related data in Table 5 to facilitate a fair comparison of the bug detection capabilities of different approaches. For example, the 3 bugs detected by Pinolo are unrelated to joins. Table 5 shows that SRS found 12, 9, 10, 20, 17, and 13 more bugs than PQS, NoREC, TLP, DQP, Pinolo, and EET, respectively. Note that the increments are calculated based

Table 5. Number of unique logic bugs triggered in 24 hours.

DBMS	PQS	NoREC	TLP	DQP	Pinolo	EET	SRS
MySQL	0	-	4	0	2	3	10
MariaDB	-	0	-	0	1	-	6
TiDB	-	-	2	0	0	0	4
PostgreSQL	0	0	0	-	-	0	2
DuckDB	-	0	1	-	-	-	1
Total	0	0	7	0	3	3	23
Increment	12	9	10	20	17	13	-

"-" indicates that the tool does not support the DBMS.
"Increments" are calculated only for commonly supported DBMSs.

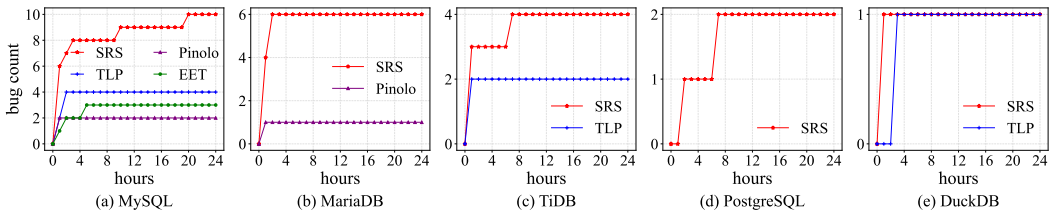

Fig. 5. Comparison of logic bug detection efficiency between SRS and the baselines. Subfigures (a)–(e) show the number of unique logic bugs discovered within 24 hours for each DBMS. Results for baselines that do not support a given DBMS are omitted, and cases where the baselines found no bugs are not shown.

only on the DBMSs commonly supported by both approaches. The other approaches were less effective in our experiments because we tested the latest versions of the DBMSs. Therefore, their reduced effectiveness suggests that discovering additional bugs has become more challenging for them, and the bugs detected by our approach are likely beyond the reach of existing approaches.

Results of efficiency. Figure 5 illustrates the logic bug detection progress over time for SRS and the baselines. Results from baseline approaches are excluded if they pertain to unsupported DBMSs or if the approaches failed to detect any bugs. At each time point, SRS consistently detects an equal or greater number of unique logic bugs compared to all baselines, highlighting its effectiveness in identifying bugs both faster and more comprehensively.

Code coverage is a commonly used metric in testing; however, it is mainly influenced by the test case generation strategy, which is not the focus of this work. We compared the code coverage achieved by these approaches, and the results showed that the test oracles implemented on SQLancer— PQS, NoREC, TLP, DQP, and SRS—achieved similar code coverage. For instance, when testing MySQL for 24 hours, PQS, TLP, DQP, and SRS achieved line coverage rates of 23.35%, 23.78%, 22.37%, and 24.19%, respectively. This similarity arises because these methods adopt comparable test case generation strategies, and the generated queries exercise similar language features.

5.4.2 Comparison to Other DBMS Logic Bug Testing Oracles. We conducted a best-effort manual analysis on the bug-inducing test cases to determine whether the logic bugs identified by SRS, as shown in Table 3, could theoretically be detected by other approaches' oracle, and whether the join-related logic bugs detected by other approaches could be identified by SRS's oracle. A test oracle is valuable as long as it can detect logic bugs that other test oracles cannot. And this is a widely used method for comparing different test oracles [23]. In this experiment, to determine whether the bugs detected by SRS could be identified by other approaches' oracle, we selected TQS/DQP,

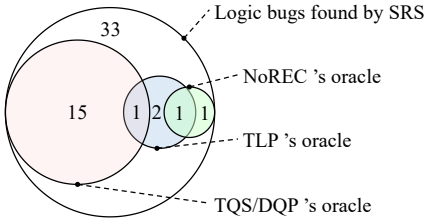

Fig. 6. Results of applying other oracles on logic bugs detected by SRS. Of the total 33 logic bugs detected by SRS, 13 were undetectable by other oracles.

TLP, and NoREC for comparison because their fixed transformation patterns allow for reliable manual validation. We excluded EET, Pinolo, and PQS due to their vast transformation spaces, which make manual validation infeasible. Specifically, EET constructs equivalent queries by adding expressions that always evaluate to `TRUE` or `FALSE`. However, the search space for such expressions is extremely large, making manual generation infeasible. Similarly, Pinolo and PQS were excluded due to scalability issues: in Pinolo, the space of approximate queries is prohibitively large, and in PQS, there can be infinitely many possible pivoted queries for a given row. To determine whether SRS can detect join-related logic bugs identified by other approaches, we collected bug reports from baselines, extracted those involving join operations, and then applied SRS's test oracle to these bug-inducing test cases to assess whether SRS could detect the bug.

Results of other oracles on bugs detected by SRS. The results are presented in Figure 6. Out of the 33 logic bugs detected by SRS, 16 can be detected by TQS/DQP's oracle; 4 can be detected by TLP's oracle; 2 can be detected by NoREC's oracle; and 13 are uniquely detected by SRS.

For TQS/DQP's oracle, we incorporated query hints and relevant system variables on the bug-inducing test cases. The result shows that 17 bugs are undetectable by TQS/DQP's oracle. When TQS/DQP fail to influence the optimizer's behavior, or when the logic bugs lie in the join implementation unrelated to optimization, TQS/DQP is unable to detect certain issues. For the example shown in Listing 2, despite applying all optimizer switches[5] and query hints[6] of MySQL, the bug remained undetectable. The inconsistency between the intersection of a left outer join and a right outer join with an inner join helps SRS detect this bug.

Although the analysis shows that 16 out of the 33 logic bugs detected by SRS could theoretically be detected by DQP's oracle, the tool of DQP failed to uncover any of them during 24 hours, as shown in Table 5. To determine why a gap exists between theoretical (Figure 6) and practical (Table 5) bug detection in DQP, we checked the join queries generated by DQP's tool, and found it supports limited join operations (i.e., cross/inner/outer/natural join). In contrast, SRS supports a broader range of join types, including semi joins and anti joins, even though these join types are not explicitly defined in the query—for example, as illustrated in Listing 4.

For TLP's oracle, we manually constructed three follow-up join queries based on the condition in the bug-inducing query, corresponding to the forms `p`, `NOT p`, and `p IS NULL`. We then examined whether the union of the results from these three queries covered all relevant relations. Our analysis shows that four of the bugs detected by SRS can also be identified by TLP's oracle, whereas the remaining logic bugs cannot be detected by it. For NoREC's oracle, we manually constructed two equivalent queries by leveraging the equivalence between the `WHERE` and `SELECT` clauses in the bug-inducing query. Our analysis shows that two of the bugs detected by SRS can also be identified by NoREC's oracle, while the remaining logic bugs cannot be detected by it. Given that neither

[5]https://dev.mysql.com/doc/refman/8.4/en/switchable-optimizations.html
[6]https://dev.mysql.com/doc/refman/8.4/en/optimizer-hints.html

NoREC nor TLP's oracle is designed to test the implementation of join operations, it is unsurprising that they fail to detect most of these logic bugs.

Listing 5. Example of applying the SRS oracle to a join-related bug from other tools' bug list

```
-- A bug-inducing case from PQS's bug list: https://www.manuelrigger.at/dbms-bugs/
CREATE TABLE t0(c0 INT);
CREATE TABLE t1(c0 INT) ENGINE = MEMORY;
INSERT INTO t0(c0) VALUES(0);
INSERT INTO t1(c0) VALUES(-1);
 -- PQS's oracle: expected: row is fetched, actual: no row is fetched
SELECT * FROM t0, t1 WHERE (CAST(t1.c0 AS UNSIGNED)) > (IFNULL("u", t0.c0));

-- SRS's oracle: "{0, -1} != {0, null} intersect {0, -1}", violating rule R06 in Table 1
SELECT * FROM t0 INNER JOIN t1 ON (CAST(t1.c0 AS UNSIGNED)) <= (IFNULL("u", t0.c0)); -- {0, -1}
SELECT * FROM t0 LEFT JOIN t1 ON (CAST(t1.c0 AS UNSIGNED)) <= (IFNULL("u", t0.c0)); -- {0, null}
SELECT * FROM t0 RIGHT JOIN t1 ON (CAST(t1.c0 AS UNSIGNED)) <= (IFNULL("u", t0.c0)); -- {0, -1}
```

Results of SRS oracle on bugs detected by other tools. We collected a total of 75 logic bugs involving join operations from the bug lists of baselines. We then applied the SRS's oracle to these bug cases and successfully reproduced 69 of them, including 16 out of 18 in TQS, 14 out of 15 in DQP, 5 out of 6 in PQS, all 5 in NoREC, 16 out of 17 in TLP, 7 out of 8 in Pinolo, and all 6 in EET. Listing 5 demonstrates the SRS oracle applied to a join-related bug from PQS's bug list. The example shows a violation of rule R06 in Table 1, where the intersection of a left and right outer join's results is not equivalent to the inner join's result, thereby illustrating SRS's effectiveness. This demonstrates the effectiveness of SRS in identifying join-related issues. The remaining three undetectable bugs involve queries that include join operations; however, their root causes are unrelated to joins. For example, one bug was caused by an incorrect result from the `MAX()` function used in the `HAVING` clause. Although the bug can only be triggered when two tables are joined, its root cause is not related to the join itself, and thus SRS cannot detect it.

Listing 6. A false positive in MySQL caused by random case variations in returned values

```
CREATE TABLE t0(c0 CHAR(1));
CREATE TABLE t1(c1 CHAR(1));
INSERT INTO t0(c0) VALUES('A'),('a');
INSERT INTO t1(c1) VALUES('a'),('b');
SELECT DISTINCT t0.c0 FROM t0; -- {A}
SELECT DISTINCT t0.c0 FROM t0 INNER JOIN t1 ON c0 != c1; -- {a}
```

6 Discussion

False alarms. The presence of non-deterministic syntactic features in DBMSs introduces false positives for our approach. However, how to effectively support such non-deterministic behavior in metamorphic testing remains an open research question. During query generation, we strive to avoid common language features with non-deterministic behavior. However, due to syntactic differences among various DBMSs, it is still possible for rare non-deterministic features to be inadvertently included. Listing 6 illustrates one such case we encountered. MySQL exhibits non-deterministic behavior when handling `DISTINCT` under a case-insensitive collation. The query `SELECT DISTINCT t0.c0 FROM t0 INNER JOIN t1 ON c0 != c1` returns `a`, which contradicts the result of `SELECT DISTINCT t0.c0 FROM t0`, namely `A`. The issue arises due to MySQL's default case-insensitive collation, which treats `A` and `a` as equivalent, leading to non-deterministic results. To avoid this false positive, we enforce case-sensitive string comparisons in MySQL. Currently, SRS filters out all language features with non-deterministic behavior, thereby eliminating any false positives.

SRS generalizability. SRS tests for logic bugs in join implementation within DBMSs through the synthesis of set relations. However, due to the complexity of join operations, enumerating all possible rules for set relation synthesis is infeasible. In this work, we design 12 representative rules that are widely used in industry, while acknowledging that additional rules could further enhance the approach. In the future, we will further explore set relation synthesis rules.

SRS scope. SRS is also capable of identifying logic bugs that occur outside the scope of join operations. Since the output of a join operation is often used as input to other language features within a query, any change in the join results can affect these inputs, potentially causing the DBMS to execute subsequent language features along different code paths. Given that these language features are designed to retrieve data satisfying specific conditions, the final query results should consistently reflect the underlying set relations among the various join operations. Ultimately, the set relations constructed through join operation transformations can serve as a bridge to test other language features. By checking whether the query results conform to or reflect these set relations, we can identify logic bugs in join operations and other language features. We received feedback from at least two logic bug reports indicating that the bug exists in language features beyond joins. Besides, if both query variants produce the same incorrect result, like all metamorphic testing tools, SRS may fail to detect certain logic bugs in a single test iteration. Nevertheless, our approach uses the set relation among join operations to construct oracle queries, which naturally perform cross-checking. Thus, SRS may expose these bugs across multiple test iterations.

The implementation beyond SQLancer. We made several modifications to SQLancer to enhance the diversity and effectiveness of the generated test cases. Specifically, we adjusted SQLancer to produce schemas with a wider range of data types (e.g., composite types) and to generate more boundary values, such as maximum and minimum values, nulls, and special characters. These boundary values are particularly effective for exposing latent bugs in DBMSs. For instance, one logic bug shown in Figure 1 could only be triggered by such boundary values, not by common inputs. To assess the impact of these enhancements, we conducted an ablation study by disabling them. Within a 24-hour testing period, SRS without these strategies detected 3 fewer logic bugs compared to the 23 bugs found by the full version, as shown in Table 5.

7 Related Work

In this section, we focus on several DBMS testing tasks and highlight how they differ from SRS.

Metamorphic testing of DBMSs. Metamorphic testing [6] addresses the test oracle problem by generating new inputs based on an existing input-output pair, inferring the expected result through metamorphic relations. Non-optimizing Reference Engine (NoREC) [23] assumes that a predicate should evaluate to the same value in both the `WHERE` and `SELECT` clauses, and leverages this assumption to construct a non-optimizable form of the query. The original and transformed queries are expected to produce consistent results. Differential Query Execution (DQE) [27] adopts a similar idea to NoREC, assuming that a predicate should yield consistent results, regardless of whether it appears in a `SELECT`, `UPDATE`, or `DELETE` statement. Equivalent Expression Transformation (EET) [13] introduces tautologies and contradictions into expressions, ensuring that the transformed query produces consistent results with the original one. Pinolo [11] transforms predicates to make them either more permissive or more restrictive, thereby generating over-approximations or under-approximations of the original query. CODDTest [32] leverages constant folding and propagation on expressions to produce equivalent queries.

Ternary Logic Partitioning (TLP) [24] assumes that a predicate `p` can evaluate to `TRUE`, `FALSE`, or `NULL`. Based on this, a query can be decomposed into three follow-up queries that return the results corresponding to `p`, `NOT p`, and `p IS NULL`, respectively. The union of these follow-up query results should be equivalent to the result of the original query. Although both TLP and SRS use the UNION

set relation as their test oracle, TLP and SRS differ in target and methodology: (1) SRS focuses on issues related to the implementation bugs of join operations, while TLP targets bugs in predicate handling code via query decomposition. (2) SRS is a mathematically rigorous framework based on four core set relations (i.e., Equivalence, Intersection, Disjointness, Union) to formally characterize join behavior. TLP decomposes a query into 3 based on predicates with `TRUE/FALSE/NULL` results.

Transformed Query Synthesis (TQS) [28] and Differential Query Plan (DQP) [3] target logic bugs in join optimizations and leverage equivalence relationships among different query plans generated for the same query. SRS is orthogonal to TQS and DQP in targets and methods: (1) TQS and DQP target join optimization bugs, while SRS focuses on join implementation bugs. Thus, SRS may also detect implementation bugs unrelated to optimization. (2) TQS and DQP construct test oracles by using query hints to execute different plans, while SRS generates test oracles by transforming the order, type, and conditions of join operations to synthesize set relations. However, not all join operations can trigger the generation of multiple query plans. As a result, some join-related bugs detected by SRS cannot be uncovered by these two approaches.

Differential testing of DBMSs. Differential testing [19] involves executing the same input on multiple versions of a system or on comparable systems, with the expectation that they produce consistent outputs. Any discrepancies observed may indicate the presence of logic bugs. RAGS [26] was one of the earliest applications of differential testing to DBMSs, identifying bugs by comparing query results across different systems. Gu et al. [10] extended this approach by introducing the use of options and hints to generate diverse query plans, enabling the evaluation of optimizer accuracy based on estimated costs. Jung et al. [14] proposed the APOLLO system to detect performance regression bugs by executing SQL queries on both older and newer versions of a DBMS.

Testing other aspects of DBMSs. Many DBMS testing techniques focus on identifying security-relevant bugs, such as memory errors. For instance, SQLSmith [1] and DynSQL [12] use grammar-based approaches to generate test cases aimed at detecting memory errors. Inspired by grey-box fuzzers like AFL [31], Squirrel [33] employs code coverage as a guide to uncover memory issues, intelligently generating inputs to explore more code paths. In addition to security issues, performance issues in DBMSs have also been a major focus of research. Amoeba [17] constructs semantically equivalent query pairs and compares their response times to find performance bugs. Cardinality Estimation Restriction Testing (CERT) [2] is designed to detect performance issues from the perspective of cardinality estimation. Given a query, CERT generates a more restrictive variant under the assumption that the cardinality estimator should predict a lower number of rows returned compared to the original query. Jung et al. [14] proposed the APOLLO system, which leverages differential testing to detect performance regression bugs in DBMSs.

Database and SQL query generation. Multiple prior works have focused on automatically generating database states and SQL queries, serving as complementary approaches to our research. Gray et al. [9] introduced a set of techniques designed to efficiently generate billions of database records. Data Generation Language (DGL) [5] is a domain-specific language for generating data exhibiting complex correlations within and across tables. Query-aware database state generation has gradually attracted the attention of researchers. QAGen [4] leverages symbolic execution to generate database states that satisfy given queries and constraints, thereby producing specific query results. ADUSA [15] employs a constraint solver to synthesize database instances and their expected outputs for a specified query and database schema. Automatic query generation is also a key component of automated testing approaches. SQLsmith [1], inspired by Csmith [30], is an open-source tool for randomly generating SQL queries. SQLRight [16] performs SQL mutation guided by code coverage feedback to explore a wider range of execution paths in DBMSs. Squirrel [33] also adopts a coverage-guided mutation approach, ensuring syntactic correctness by performing mutations on the intermediate representation.

8 Conclusion

In this paper, we presented SRS to detect logic bugs of join implementation in DBMSs by leveraging set relations among different join operations. Our insight is that different joins involve various join types, orders, and constraints, which are not mutually independent but instead follow specific set relations. SRS thoroughly tests join operations by transforming join operations within queries, and then detects logic bugs by checking whether the resulting join sets satisfy specific set relations. We implemented SRS and evaluated it across five widely-used DBMSs, discovering 36 previously unknown bugs, all of which have been confirmed. Furthermore, the comparison with state-of-the-art methods showed that SRS can effectively detect logic bugs that those approaches fail to identify. We believe that SRS is a practical and widely applicable testing approach for detecting logic bugs in the join implementations.

Acknowledgements

We thank the shepherd and reviewers for their valuable comments. This research is sponsored by the National Key Research and Development Program of China (No.2022YFB4501703), National Natural Science Foundation of China (No.62572222), and Jiangxi Province Graduate Innovation Special Fund Project (YC2024-B031).

Data Availability

The artifact of SRS is available at https://anonymous.4open.science/r/SRS-9310

References

[1] Bo Tang Andreas Seltenreich and Sjoerd Mullender. 2018. SQLsmith: a random SQL query generator. https://github.com/anse1/sqlsmith.git

[2] Jinsheng Ba and Manuel Rigger. 2024. CERT: Finding Performance Issues in Database Systems Through the Lens of Cardinality Estimation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 133, 13 pages. doi:10.1145/3597503.3639076

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

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

[5] Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible database generators. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 1097–1107.

[6] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–27.

[7] E. F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (jun 1970), 377–387. doi:10.1145/362384.362685

[8] DuckDB. 2025. https://duckdb.org/

[9] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, USA) *(SIGMOD '94)*. Association for Computing Machinery, New York, NY, USA, 243–252. doi:10.1145/191839.191886

[10] Zhongxian Gu, Mohamed A. Soliman, and Florian M. Waas. 2012. Testing the accuracy of query optimizers. In *Proceedings of the Fifth International Workshop on Testing Database Systems* (Scottsdale, Arizona) *(DBTest '12)*. Association for Computing Machinery, New York, NY, USA, Article 11, 6 pages. doi:10.1145/2304510.2304525

[11] 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.

[12] 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 (USENIX Security 23)*. USENIX

Association, Anaheim, CA, 4949–4965. https://www.usenix.org/conference/usenixsecurity23/presentation/jiang-zu-ming
[13] Zu-Ming Jiang and Zhendong Su. 2024. Detecting logic bugs in database engines via equivalent expression transformation. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 44, 15 pages.
[14] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: automatic detection and diagnosis of performance regressions in database systems. *Proc. VLDB Endow.* 13, 1 (sep 2019), 57–70. doi:10.14778/3357377.3357382
[15] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O Laleye, and Sarfraz Khurshid. 2008. Query-Aware Test Generation Using a Relational Constraint Solver. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. IEEE Computer Society, USA, 238–247. doi:10.1109/ASE.2008.34
[16] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4309–4326. https://www.usenix.org/conference/usenixsecurity22/presentation/liang
[17] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 225–236. doi:10.1145/3510003.3510093
[18] MariaDB. 2025. https://mariadb.org/
[19] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.
[20] Priti Mishra and Margaret H. Eich. 1992. Join processing in relational databases. *ACM Comput. Surv.* 24, 1 (March 1992), 63–113. doi:10.1145/128762.128764
[21] MySQL. 2025. https://www.mysql.com/
[22] PostgreSQL. 2025. https://www.postgresql.org/
[23] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1140–1152. doi:10.1145/3368089.3409710
[24] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (nov 2020), 30 pages. doi:10.1145/3428279
[25] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 667–682. https://www.usenix.org/conference/osdi20/presentation/rigger
[26] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. Citeseer, 618–622.
[27] 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)*. 2072–2084. doi:10.1109/ICSE48619.2023.00175
[28] 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.
[29] TiDB. 2025. https://www.pingcap.com/
[30] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532
[31] Michal Zalewski. 2015. AFL: American Fuzzy Lop. https://github.com/google/AFL
[32] Chi Zhang and Manuel Rigger. 2025. Constant Optimization Driven Database System Testing. *Proc. ACM Manag. Data* 3, 1, Article 24 (Feb. 2025), 24 pages. doi:10.1145/3709674
[33] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 955–970.

Received April 2025; revised July 2025; accepted August 2025