# ARG: Testing Query Rewriters via Abstract Rule Guided Fuzzing

Dawei Li[1,†], Yuxiao Guo[1,†], Qifan Liu[1], Jie Liang[1,✉], Zhiyong Wu[2], Jingzhou Fu[2], Chi Zhang[2], Yu Jiang[2]

[1]Beihang University, Beijing, China
[2]KLISS, BNRist, School of Software, Tsinghua University, Beijing, China

***Abstract*—Query rewriters transform a query into a more efficient yet semantically equivalent form, which is vital for optimizing query execution. Despite its importance, query rewriting is inherently complex, influenced by factors including rewrite rule design, rule interactions, and semantic preservation. Consequently, its implementation struggles to prevent problems, which may result in system crashes or incorrect query results. Existing DBMS testing approaches are generally designed for broad bug detection. However, due to the diversity of rewrite rules, they cover only a limited subset of rewrite scenarios, potentially overlooking critical bugs.**

**In this paper, we propose Abstract Rule Guided (ARG) fuzzing to detect bugs in query rewrites. The key idea is to use feedback from abstract rules to guide query generation, thereby activating more rewriting logic and enhancing bug detection. Abstract rules provide a unified representation of the patterns (e.g., AST structures and related constraints) that trigger rewrites, as well as the resulting transformations. We track abstract rules to identify which patterns have been covered. This feedback is then used to dynamically adjust query generation, prioritizing unexplored patterns to avoid redundancy and expose more rewriting logic. We implemented ARG to test four popular query rewrites, namely Apache Calcite, WeTune, SQLSolver, and LearnedRewrite. ARG discovered 38 previously unknown bugs, consisting of 4 crashes, 13 invalid SQL outputs, and 21 semantic deviations. Among them, 19 have been confirmed, while the remaining cases are still under investigation. We also compared ARG against popular DBMS testing tools. In 24 hours, ARG triggered 76% and 1017% more written rules, triggered 13 and 15 more bugs than SQLsmith and SQLancer, respectively.**

***Index Terms*—Query Rewriter, Rule Feedback, Bug Detection**

## I. Introduction

Query rewriting is a widely adopted and critically important technique in database management systems (DBMSs) [4], [28]. Many DBMSs employ query rewriters as the logical optimization phase of the query optimization, leveraging equivalence rules to transform complex query statements into more efficient yet semantically equivalent forms [7], [29], [32]. These query rewriters enhance query performance, leading to substantial savings in computational resource costs for enterprises and generating considerable economic benefits.

In practice, query rewriting is inherently complex and variable, requiring the restructuring of query expressions at the logical level while preserving semantic equivalence. This process demands not only an accurate understanding of contextual information, but also comprehensive consideration of factors such as data distribution and index availability. Additionally, variations in query languages, nested structures, and evolving business rules further complicate the task. Consequently, rewriter implementations often struggle to prevent bugs: they may occasionally crash during execution. In more complex scenarios, rewritten queries can be invalid, containing syntax or semantic errors, or violate semantic equivalence, resulting in incorrect results.

For example, Figure 1 illustrates a bug in Apache Calcite [3], where the rewritten query produces results that are not semantically equivalent to the original. Specifically, Apache Calcite mistakenly assumes certain columns are uniquely identifying, which leads to incorrect pruning of result sets containing `NULL` values, thereby violating query semantics. This bug is triggered under two conditions. First, when the query involves a self-join, the rewriter may reorder projected columns incorrectly, causing essential expressions to be lost. Second, if the primary key is of type `FLOAT`, unintended type conversions during rewriting can introduce precision errors.

```
-- Cast operation leads to unequal results
CREATE TABLE t1 (c1 float NOT NULL, PRIMARY KEY(c1));
INSERT INTO t1 VALUES (0), (0.0963786);
-- Original query
SELECT t0.c1 AS c0, t2.c1 AS c1 FROM t1 AS t0
   RIGHT JOIN t1 AS t2 ON TRUE WHERE t0.c1 IS NOT NULL;
-- Result:
   {0, 0.0963786}, {0, 0},
   {0.0963786, 0.0963786}, {0.0963786, 0}
-- Rewritten query
SELECT CAST(t1.c1 AS REAL) AS c0, t10.c1 FROM t1, t1 AS t10;
-- Result:
   {0.09637860208749771, 0}, {0, 0},
   {0.09637860208749771,0.0963786}, {0, 0.0963786}
```

Fig. 1. **The query to trigger a rewrite bug in Apache Calcite.** First, the order of projection operations is incorrectly adjusted. Moreover, an extraneous `CAST` operation is added to the float-type column `c0`, converting it to `DOUBLE` with altered precision. These issues cause nonequivalence.

**However, there is currently a lack of systematic testing methodologies tailored specifically for query rewrites.** Many existing DBMS testing tools [22], [23], [31] effectively uncover logic bugs and crashes, but they are primarily designed for broad bug detection rather than targeting query rewrites. Lacking rewriting semantic awareness, they often fail

[†]These authors contributed equally to this work and should be considered co-first authors.
[✉]Jie Liang is the corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00151

to systematically trigger complex rule combinations, potentially missing critical bugs in the rewriting logic. Moreover, due to the diversity and implementation-specific nature of rewriting rules, these methods can cover only a limited subset of rewriting scenarios. For example, SQLsmith [23] generates complex queries based on fixed grammar rules to enhance syntactic and semantic correctness; nevertheless, many queries remain structurally similar and lack the diversity needed to fully exercise the breadth of rewriting logic.

In this paper, we propose **A**bstract **R**ule **G**uided (ARG) fuzzing to detect bugs in query rewrites. *The key idea is to guide future query generation using feedback abstracted from the rewriting rules.* Rewriting rules specify the transformation relationships between the original and the rewritten query, serving as the foundation of the query rewriter [26]. By abstracting these rules, ARG captures high-level patterns (e.g., changes in query structure) of rewriting behavior. By recording these abstract rules, ARG identifies untriggered patterns and guides the query generator to construct new queries containing them, thereby exploring previously unvisited rewriting paths. However, it has to face the following challenges:

(1) *The unified abstraction of diverse rewriting rules*. The rewriting rules vary significantly in structure, semantics, and context, often involving implicit conditions. The complexity makes it difficult to develop a representation that is both expressive and practical.

(2) *Efficient query generation guided by these abstractions*. It requires precise mapping between query structures and rules, as well as tracking which patterns have been tested. Moreover, balancing the exploration of new patterns with the exploitation of known triggers is essential.

ARG addresses the first challenge by extracting structural features from original and rewritten queries, along with schema constraints, to create a unified triplet representation. Specifically, ARG leverages abstract syntax trees (ASTs) to capture structural differences between queries, forming a triplet-based abstract rule that includes the pre-rewrite AST subtree, the post-rewrite subtree, and encoded constraints such as primary and foreign keys. To address the second challenge, ARG introduces a feedback-driven query generation mechanism. It maintains a set to record all abstract rules it finds. Based on that, the SQL generator probabilistically explores alternative AST structures to activate additional rewriting rules. Furthermore, different database schemas may trigger different behaviors in the query rewriter. Therefore, ARG actively modifies the database schemas and reuses existing abstract rules to generate queries, enhancing the activation of diverse rewriting behaviors. To detect bugs, ARG continuously monitors for rewriter crashes, validates query results for syntax and semantic correctness, then executes and compares original and rewritten queries to detect mismatches.

We implemented ARG to test four popular query rewrites, namely Apache Calcite [3], WeTune [26], SQLSolver [5], and LearnedRewrite [33], [34]. ARG discovered 38 previously unknown bugs across these query rewriters. Among them, 19 have been confirmed by the rewriter developers, while the remaining cases are still under investigation. The detected bugs result in three major and severe failure symptoms: 4 system crashes, 13 invalid SQL outputs, and 21 query semantic deviations, which causes database service interruptions, execution failures, or incorrect query outputs, respectively. We also compared ARG against popular DBMS testing tools. In 24 hour experiment on these query rewriters, ARG triggered 76% and 1017% more written rules, triggered 13 and 15 more bugs than SQLsmith and SQLancer, respectively, In summary, we make the following contributions:

- We observe that while query rewriters are widely used in DBMSs, bugs persist and can cause serious issues, yet effective testing tools remain lacking.
- We propose ARG, the first fuzzing-based approach for query rewriters, which extracts abstract rules to guide query generation and trigger more rewriting behaviors.
- We uncovered 38 unique bugs in popular query rewriters like Apache Calcite, including 4 crashes, 13 invalid SQL outputs, and 21 query semantic deviations.

## II. Background and Motivation

**Query Rewriter.** Query rewriting plays a fundamental role in database query optimization. It transforms initial SQL queries into semantically equivalent but more efficient forms, whether integrated into DBMSs or implemented as standalone components. Rooted in relational algebra, it restructures queries at the logical level to optimize execution while preserving their original semantics. Most rewriters rely on predefined equivalence rules and dynamic pattern-matching techniques to identify and apply valid transformations.

The implementation of rewriters is inherently complex. Ensuring that the system accurately interprets the underlying intent of each query, despite variations in phrasing or structure, requires sophisticated SQL understanding capabilities. This complexity is further compounded by the necessity to handle ambiguous or incomplete queries gracefully, necessitating robust disambiguation mechanisms and context-aware inference to deliver consistent and reliable results. Moreover, implementations differ across DBMSs, ranging from proprietary solutions like MySQL Rewrite Plugin [1] to modular open-source alternatives such as Apache Calcite [3], which is widely recognized for its architectural flexibility.

**Rewriting Rules.** Query rewriters depend on equivalence rules, which define the transformation relationships between the original query and the rewritten query [17]. Different DBMSs adopt varying representations for these rules. For example, MySQL Rewrite Plugin uses system tables to express rules, Apache Calcite designs RelOptRule objects, while WeTune employs text-based representations.

Despite diverse forms of statement, these rules can fundamentally be summarized as combinations of three elements: original structure, rewritten structure, and constraint conditions. During the rewriting process, the rewriter first matches the overall structure or partial fragments of the query statement with the original structures defined in the rules. When matching rules are found, the rewriter reassembles the matched

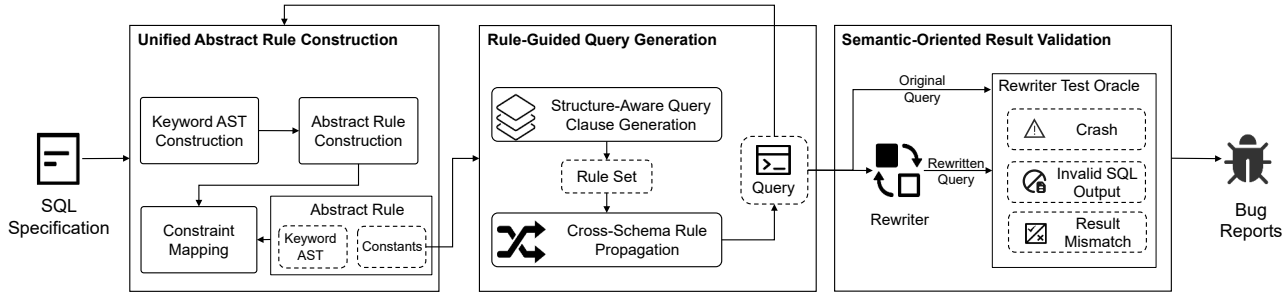


Fig. 2. **Design of ARG.** ARG tests the query rewriter through the following process: (1) Unified Abstract Rule Construction. ARG constructs abstract rules by extracting keyword ASTs from both original and rewritten queries, locating structural differences as rule pairs, and combining them with constraint information to capture semantic context. (2) Rule-Guided Query Generation. ARG uses abstract rules as feedback to guide the SQL generator in producing structurally diverse and targeted queries. By encoding rule features as bitmaps and reusing generator states, it dynamically adapts syntax generation and enables cross-schema rule propagation to improve rule coverage. (3) Semantic-Oriented Result Validation. ARG continuously monitors for rewriter crashes, checks each query result for validity (free of syntax or semantic errors), then executes both queries and compares their results to identify mismatches.

statement portions into corresponding target structures. If the initially rewritten query subsequently matches other rules, the rewriter performs iterative rewriting through equivalent rule chaining, enabling chained application of equivalence rules.

**Limitations of Current Tools for Query Rewriter Testing.** As recent studies have shown [9], ensuring a bug-free implementation of query rewriters in DBMSs remains a significant challenge. Despite their importance, there is currently a lack of systematic testing methodologies specifically designed for query rewriters. While current DBMS fuzzers have proven effective for general database testing, they fall short when applied to query rewriters due to their limited understanding of internal rewriting logic and rules.

Specifically, existing tools often lack a deep understanding of rewriting rules and the internal rewriting logic, making it difficult to identify and trigger precise rewriting scenarios. Traditional generation-based fuzzers like SQLancer and SQLsmith create diverse SQL statements based on syntax rules but struggle to systematically activate specific rewriting rule combinations due to their random nature. Mutation-based tools rely on instrumenting the entire DBMS to gather coverage and runtime information, which is impractical for query rewriters that operate independently in isolated environments. As a result, these tools cannot effectively guide testing for rewriters. Therefore, traditional random testing methods fail to reliably cover complex rewriting paths, limiting their ability to detect bugs and verify query rewriter correctness.

**Basic Idea of ARG.** *The key idea of* ARG *is to use feedback from abstract rules to guide query generation, thereby activating more rewriting logic and enhancing bug detection.* ARG is designed in accordance with the operational workflow of mainstream query rewriters. By analyzing the structural features of keywords in queries before and after rewriting, ARG uniformly represents rewriting rules, enabling it to capture rule information throughout the rewriting process and characterize rewriting behavior. Using this feedback, ARG guides the SQL generator to produce structurally diverse SQL statements, facilitating more comprehensive activation of equivalence rewriting rules. This method achieves rewriting awareness without requiring knowledge of the internal implementation details, enabling more targeted and effective testing of rewriting logic.

## III. Design

Figure 2 illustrates the architecture of ARG, which operates through three core steps: Unified Abstract Rule Construction, Rule-Guided Query Generation, and Semantic-Oriented Result Validation. In Unified Abstract Rule Construction, ARG extracts syntactic structures and constraint information from the original and rewritten queries, identifies structural differences as rule pairs, and integrates constraint information to form triplet-based abstract rules. In Rule-Guided Query Generation, ARG uses abstract rules to guide the SQL generator in exploring syntax variations. It also adapts database schemas dynamically, reusing abstract rules across schema changes to enhance rule coverage. In Semantic-Oriented Result Validation, ARG continuously monitors the query rewriter's execution for crashes. It also validates the rewritten queries by checking for syntax and semantic correctness. Then, ARG runs both the original and rewritten queries, comparing their execution outcomes. Any crash, invalid output, or semantic deviation will be flagged as a potential rewriting bug.

### A. Unified Abstract Rule Construction

Although various query rewriters exhibit significant differences in grammatical rule definition and implementation forms, their core rule-matching mechanisms fundamentally rely on structured feature analysis of original query statements and schema-matching processes. Therefore, by leveraging the structural feature differences between original and rewritten queries, combined with the constraints imposed by the target database schema, we implement a black-box unified abstract construction to express equivalence rules.

**Unified Representation of Abstract Rules.** We employ abstract rules to uniformly represent both individual rules

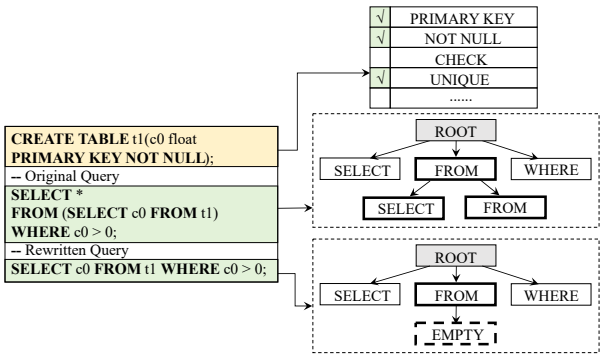


Fig. 3. **The Process of Abstract Rule Reconstruction.** The abstract rule reconstruction involves three core processes: (1) Constructing multi-branch keyword ASTs $(T_o, T_r)$ by parsing the grammatical hierarchies of the original query and rewritten query. (2) Extracting rule pairs $(F_o, F_r)$ through node-hash synchronization to locate divergent syntax subtrees. (3) Establishing a unified constraint bitmap through automatic encoding of constraints.

and their rule-chaining combinations across different rewriters, where rule-chaining combination specifically refers to the process where rewritten queries generated by prior rules are dynamically fed into subsequent rules for continuous processing. We define the structure of an abstract rule as a triple $\mathcal{R} = (F_o, F_r, C)$, where $F_o$ and $F_r$ are tree structure fragments composed of keywords in the original query and rewritten query, respectively, and $C$ is a constraint set under the table structure corresponding to $F_o$. The rule denotes that when constraint $C$ is satisfied, query statements or subclauses matching the $F_o$ structure can be rewritten into query statements or subclauses corresponding to the $F_r$ structure.

As shown in Figure 3, the construction of abstract rules involves three processes. (1) Keyword AST Extraction. By parsing the grammatical structures of the original query and rewritten query, they are converted into hierarchical tree representations. During this process, SQL keywords are mapped as multi-branch tree nodes, and nested subqueries are processed through recursion by dynamically linking sub-grammatical subtrees to main structure nodes. At this stage, corresponding keyword ASTs are constructed for the original query and rewritten query, respectively, denoted as $T_o$ and $T_r$.

(2) Rule Extraction. First, hash-based feature encoding is performed on nodes of the original keyword tree $T_o$ and rewritten keyword tree $T_r$, converting node types and their immediate child node sequences into unique hash values, which forms the basis for difference localization. Then, through top-down recursive traversal for synchronized comparison of node hash values between $To$ and $Tr$, the process locates tree segments when detecting the first hash mismatched node. It thereby defines the subtrees rooted at the mismatched positions in $T_o$ and $T_r$ as $F_o$ and $F_r$, respectively, extracting the $(F_o, F_r)$ rule pair. As shown in Figure 3, $F_o$ consists of the `FROM` node and its subquery nodes (`SELECT` and `FROM`) within the `FROM` clause, while $F_r$ contains only a single FROM node. Due to potential multi-layered structural differences in syntax trees, this process may extract multiple $(F_o, F_r)$rule pairs from a single pair $(T_o, T_r)$.

(3) Constraint mapping. Based on the database table structures involved in the original query, we automatically construct a dynamic constraint bitmap. When the source query involves single-table retrieval, we encode the primary keys, foreign keys, and other constraint conditions of the corresponding table in a bitmap format. As illustrated in Figure 3, the `PRIMARY KEY` and `NOT NULL` constraints are extracted from `CREATE` statements and mapped into a bitmap structure. For queries with multi-table join operations, we record constraint conditions from each participating table and consolidate them through bitwise operations to integrate constraints of different tables. This process ultimately generates a unified constraint feature representation model, denoted as $C$.

Figure 3 gives an example extracts abstract rules from the SQL before and after rewriting. The original query is "`SELECT * FROM (SELECT c0 FROM t1) WHERE c0 > 0;`", and the rewritten query is "`SELECT c0 FROM t1 WHERE c0 > 0;`", whose respective keyword trees are denoted as $T_o$ and $T_r$. Based on the structural changes in the AST, we extract the fragments $F_o$ and $F_r$. These fragments correspond to `FROM (SELECT FROM)` and `FROM` respectively. Therefore, the abstract rule is <`FROM (SELECT FROM)`, `FROM`, (`NOT NULL`)>.

**Deduplication of Abstract Rules.** When testing rewriters, ARG generates numerous abstract rules. To perform accurate statistical analysis of rule quantities, structural characteristics, and rule coverage validation, it is necessary to conduct uniqueness identification and deduplication of the rules. We observe that deep nesting structures (e.g., multi-level subqueries or complex conditionals) cause syntactic subtrees to diverge significantly in form. This phenomenon not only causes functionally identical rules to be counted redundantly but also reduces the accuracy of rule function identification due to detailed information carried by deep nodes. To address this issue, we propose a hierarchical hashing strategy that uniquely identifies computational rules through multi-layer node encoding. This methodology extracts hash combinations from the first three $F_o$ and $F_r$ node hierarchies as unique identifiers, enabling robust nested structure recognition while eliminating volatility from deeper layers.

ARG uses Algorithm 1 to determine whether rule $\mathcal{R}$ is a new rule. Specifically, when $\mathcal{R}$ is refactored, unique fingerprints are calculated for $F_o$ and $F_r$ in $\mathcal{R}$, resulting in $(fp_o, fp_r)$ for subsequent rule fast matching (Lines 1-2). Then, all rule fingerprints in $S$ are quickly matched against $(fp_o, fp_r)$ to filter rules in $S$ with the same structure as $\mathcal{R}$. These matching rules are stored in $match$. If no structurally identical rules are found, $\mathcal{R}$ is considered a new rule and added to the existing rule set, forming a new rule set $S'$. When structurally similar rules are found in $S$, ARG checks whether constraint $C$ differs from $rule.C$. If a difference exists, $\mathcal{R}$ is identified as a new rule and added to the new rule set (Lines 3-13). For unique identification of a rule, we design a fingerprint extraction function that extracts hash values from the first three layers of keyword AST nodes as unique identifiers, where the

**Algorithm 1:** Deduplication of Abstract Rules

**Input :** Existing Rule Set $S$.
Current Rule $\mathcal{R} = (F_o, F_r, C)$.
**Output:** New Rule Set $S'$

```
1  fp_o ← getfingerprint(R.F_o) ;
2  fp_r ← getfingerprint(R.F_r) ;
3  initiate(match) ;
4  foreach rule ∈ S do
5  |  match ← fastmatch(rule, fp_o, fp_r) ;
6  end
7  if match.state = True then
8  |  S' ← S ∪ {R} ;
9  else
10 |  C_match ← match.rule.C ;
11 |  S' ← (C_match = C) ? S : S ∪ {R} ;
12 end
13 return S' ;
14 Function getfingerprint(F):
15 |  currentHash ← 0 ;
16 |  foreach childNode ∈ F.node.children do
17 |  |  if childNode.depth < 3 then
18 |  |  |  fp_c ← getfingerprint(childNode) ;
19 |  |  |  currentHash ← (currentHash + fp_c) ;
20 |  |  else
21 |  |  |  return childNode.hash ;
22 |  |  end
23 |  end
24 |  return currentHash ;
25 End Function
```

hash value of a node is the sequential combination of the node and its direct child nodes (Lines 14-25).

### B. Rule-Guide Query Generation

Building upon the abstract rules extracted from ARG, we establish a dynamic feedback mechanism by analyzing grammatical structure patterns and constraint attributes of the abstract rules. This mechanism enables real-time regulation of the states of SQL generator to produce SQL queries with diversified structures, thereby effectively triggering the rewriter's equivalent rewriting rules. As Figure 4 shows, we employ an integrated rule feedback methodology that combines Syntax-Aware Query Adaptation and Cross-Schema Rule Propagation.

**Structure-Aware Query Generation.** Based on the structural connection between $F_o$ and $F_r$ in abstract rules and common rewrite strategies, ARG uses a feedback mechanism for test case generation with rule behavior classification and syntax awareness. Specifically, the core logic for rule behavior identification centers on the root nodes of $F_o$ and $F_r$ syntax trees. By locating position differences between $F_o$ and $F_r$ root nodes in main SQL queries or subqueries and conducting layer-by-layer comparisons between nodes, we can determine the operational scope and specific types of rewrite behaviors (e.g., keyword deletion, replacement operations). The behavior of query rewriting can be categorized into three types: operation merging, operation movement, and predicate translation [10]. Our method can effectively identify structural changes in keyword trees between original and rewritten queries, thereby detecting instances of operation merging and operation movement. For predicate translation behavior, although we cannot directly extract changes in predicate logic,

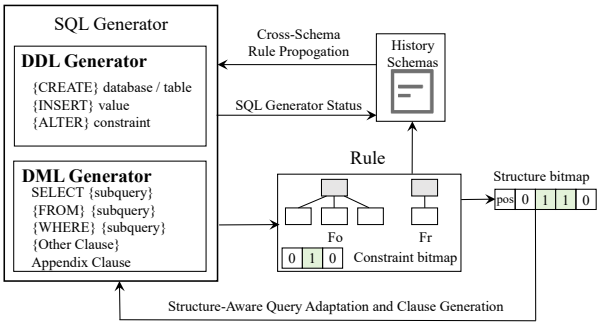


Fig. 4. **The Process of Rule-Guided Query Generation.** Rule feedback involves two processes: (1) Structure-Aware Query Generation. This process employs a bitmap data structure to characterize the functional features of abstract rules. ARG dynamically adjusts the generation probability of corresponding syntactic structures in the SQL generator through real-time analysis of these bitmap features, achieving context-sensitive adaptation of query statements. (2) Cross-Schema Rule Propagation. Upon detecting the triggering of new rules, ARG instantly records the complete contextual state of the SQL generator and applies these state features to historical schemas, thereby activating related rules across different schemas.

we can identify logic alterations caused by structural transformations, such as converting OR predicates to `IN` predicates.

Through statistical analysis of the keyword structures of abstract rules, we find that the aforementioned three types of operations can be achieved through combinations of the following four categories of rule behaviors: (1) WHERE-clause: structural adjustments to `WHERE` conditions; (2) JOIN-change: semantic transforms of table join methods; (3) Subquery-change: nested modifications of subquery structures; (4) UNION-related: logical reorganization of set operations. Simultaneously, we locate the position of the $F_o$ root node within $T_o$, denoted as $pos$, to represent the location information where transformations occur in the original query.

Based on the classified rules, ARG enables syntax-aware feedback for test case generation. Specifically, when SQL queries generated by the SQL generator are successfully mapped to abstract rules, ARG performs real-time feature matching against four predefined structure categories, generating a binary bitmap encoding. We collectively combine the $pos$ and bitmap encoding into feedback information, denoted as $Feedback = pos[x_0, x_1, x_2, x_3]$ (e.g., $Feedback = 5[1, 1, 1, 0]$ indicates the subtree rooted at the fifth keyword node in $T_o$ is rewritten with WHERE-clause, JOIN-change, and Subquery-change behaviors). This bitmap is instantly feedback to the generator, which dynamically reduces generation probabilities for existing structures in subsequent iterations, thereby exploring more syntactic combinations. For long-untriggered structure types, ARG implements a monitoring mechanism. When any bitmap position remains 0 for multiple consecutive cycles, ARG proactively adjusts generation weights to guide the generator to insert corresponding clause structures. This strategy effectively prevents test case homogenization and systematically enhances the completeness of equivalence rule coverage.

We take Figure 1 as an example to demonstrate the details of

query generation. During the Unified Abstract Rule Construction stage, ARG extracts two abstract rules <FROM(RIGHT JOIN), FROM, (NOT NULL)> and <WHERE, -, (NOT NULL)> by analyzing SQL structure differences. In the Rule-Guided Generation stage, ARG generates the encodings $2[0,1,0,0]$ and $3[1,0,0,0]$ for the two abstract rules, respectively. The indices 2 and 3 indicate the positions in the original query where these rules can be applied. ARG then randomly selects one or more of these encodings to generate new test cases. For instance, when ARG uses only the encoding $3[1,0,0,0]$, it modifies the AST structure of the `WHERE` clause. While other elements like table names and variables in the query may change, the overall keyword structure is preserved. This process can ultimately generate the following query: `SELECT * FROM t1 RIGHT JOIN t0 ON t0.c0=t1.c0 WHERE EXISTS(SELECT * FROM t0)`.

**Cross-Schema Rule Propagation.** During the test workflow of ARG, the SQL generator builds lots of different database schemas by randomly generating multiple sets of DDL statements. The rule-matching process in the query rewriter strictly relies on the semantic constraints of database schemas. Specific rules are activated to rewrite original queries only when the schema satisfies particular patterns. However, queries generated under different schemas might involve identical constraint combinations, thereby activating abstract rules with the same constraint condition. This characteristic creates potential opportunities for cross-schema rule propagation.

To enhance the capability of reconstructing abstract rules across diversified schema, we design a state feedback-based cross-schema rule propagation mechanism. Specifically, ARG dynamically monitors rule activation status during rewriter testing in specific schema. When novel rules are triggered under a particular schema, ARG records the SQL generator's core state vector $State$ and previous binary bitmap encoding $Feedback$ in real-time. The $State$ parameter influences field combinations and structural features of generated SQL statements. For new schema testing, ARG probabilistically samples ($State$, $Feedback$) pairs from historical records and substitutes them into the current testing state. By leveraging a metadata-driven approach, propagate historical states capable of triggering new rules to other schemas with similar constraints, thereby enhancing the extraction capability of abstract rules and covering a broader range of rules in the rewriter.

### C. Semantic-Oriented Result Validation

Testing query rewriters requires not only executing queries but also establishing an effective test oracle to determine the correctness of rewritten results. To this end, we adopt a Semantic-Oriented Result Validation approach, which can more accurately identify defects in the rewriter at the semantic and logical levels. Only through such validation can errors arising during query rewriting be systematically detected and classified, ensuring data accuracy and system stability.

As shown in Table I, we categorize rewriter bugs into three distinct types based on their execution behavior and result discrepancies: (1) Rewriter Crash. When executing a rewrite operation, the rewriter requires an original query and schema. If it crashes and fails to output a rewritten query, this signals a rewriter crash incident. (2) Invalid SQL Output. If the original query executes successfully and returns results, but the rewritten query fails due to semantic or syntactic errors, the DBMS will raise an invalid exception. This behavior indicates an invalid SQL output issue in the rewriter. (3) Semantic Deviation. When original and rewritten queries both execute successfully yet yield unequal results ($R(Q) \neq R(Q')$), this indicates that the rewriter has a semantic deviation problem. Semantic deviation generates logic errors without detection, silently propagating incorrect data to downstream systems. This absence of warnings critically undermines data integrity.

TABLE I
THREE VALIDATION CRITERIA AND RELATED BUG TYPE

| Rewriting | Symptom | Bug Type |
|---|---|---|
| $Q \rightarrow None$ | Runtime Crash | Rewriter Crash |
| $Q \rightarrow Q'$ | Syntactic/Semantic Error | Invalid SQL Output |
| $Q \rightarrow Q'$ | $R(Q) \neq R(Q')$ | Semantic Deviation |

To find these three types of bugs, ARG continuously monitors the query rewriter's execution for crashes that indicate critical failures. It validates whether the rewritten queries are valid by checking for syntax and semantic correctness. Then, ARG runs both the original and rewritten queries, comparing their execution outcomes and results. Any divergence, whether it is a crash, invalid output, or semantic deviation, is flagged as a potential rewriting bug.

## IV. IMPLEMENTATION

We implemented ARG with 8.4k lines of Java code and about 1k lines of C++ code, built an automated testing framework for the query rewriter, and implemented mechanisms for abstract rule extraction and rule-based feedback. The rule extraction component uses the AST node identity binding mechanism of sql-formatter [21], which identifies SQL structures across dialects and tags syntax tree nodes with feature identifiers, enabling precise keyword AST extraction.

To implement the rule feedback mechanism, we developed a customized version of SQLsmith to generate DML queries, introducing probabilistic control flags to govern its syntax structure generation strategy. To generate DDL statements and enhance abstract rule extraction, we integrated SQLancer to construct dynamic database schemas and populate test data. This combined approach mitigates the limitations of each tool: SQLancer's native FUZZER lacks sufficient coverage for complex rule patterns, while SQLsmith, despite its strength in generating intricate query structures, does not support adaptive schema generation. To integrate new query rewriters with ARG, developers only need to implement our standardized rewrite interface, which takes a query and schema as input and returns the optimized statement.

## V. Evaluation

To evaluate the effectiveness and efficiency of ARG in detecting bugs in query rewriters, we design experiments to answer the following questions:

- **Q1:** Can ARG detect bugs in real-world query rewriters?
- **Q2:** How does ARG compare with existing techniques?
- **Q3:** How does rule feedback guidance contribute to the performance of ARG?

### A. Evaluation Setup

**Test Rewriters.** To evaluate the bug detection efficiency of ARG, we tested four popular and widely used query rewriters: Apache Calcite [3], WeTune [26], SQLSolver [5], and LearnedRewrite [33], [34]. Among them, Apache Calcite is one of the most popular open-source query rewriters, while LearnedRewrite, WeTune, and SQLSolver represent three of the most recent advances in query rewriting.

**Basic setup.** The evaluation was conducted on a server running 64-bit Ubuntu 22.04.5 LTS, equipped with an AMD EPYC 7763 64-core processor (128 threads) and 500 GB of main memory. We evaluated each rewriter on its corresponding supported DBMS (e.g., MySQL), with all rewriters updated to their latest available versions at the time of testing: Apache Calcite (v1.39.0) [2], WeTune (commit 3e36def4) [27], SQLSolver (commit c03062f) [24], and LearnedRewrite (commit 4fd732b) [12]. For quantitative comparisons, we ran the Docker containers for each rewriter testing experiment (including DBMS, tested rewriters, and ARG) with 10 CPU cores and 10 GiB of main memory. To detect real-world rewriter bugs, we continuously ran ARG on tested rewriters for two weeks.

### B. Rewriter Bug Detection

ARG successfully discovered 38 previously unknown bugs in four query rewriters within two weeks, including 21 semantic deviations, 13 invalid SQL outputs, and 4 crashes. Among them, 19 have been confirmed and acknowledged.

**Statistics.** Table II summarizes the distribution of the 38 detected bugs across four representative rewriter systems. Specifically, ARG discovered 12, 11, 8, and 7 bugs in Apache Calcite, WeTune, SQLSolver, and LearnedRewrite, respectively. Among them, 19 bugs were confirmed by developers as previously unknown, while the remaining cases are still under investigation. These results demonstrate that bugs in query rewriters remain prevalent even in production-level and academically tested systems. ARG can detect such bugs by constructing diverse query variants guided by abstract rule feedback and validating semantic equivalence after rewriting.

*Impact of the Detected Bugs.* The detected bugs lead to three major behaviors, including 21 semantic deviations, 13 invalid SQL outputs, and 4 system crashes. Table II also lists the types and descriptions of these bugs. 21 bugs caused silent logic bugs, returning incorrect results without any warning, which may severely affect data integrity in downstream applications. 13 bugs caused the invalid SQL outputs, where rewritten queries contained invalid syntax, causing DBMS execution failures. 4 bugs caused the system crashes, which make the rewriter unavailable. These bugs emphasize the need for rewriter-specific testing techniques such as ARG that can target and exercise complex rewrite logic.

**Case Study.** To illustrate the characteristics and root causes of the bugs detected by ARG, we present two case studies featuring symptoms of invalid SQL output and semantic deviation.

*(1) An Invalid SQL Output in WeTune.* As shown in Figure 5, an invalid SQL output issue is uncovered in WeTune. The original query performs a `RIGHT JOIN` between table `t1` and a subquery aliased as `t2`, using the join condition `t1.c0 = t2.c0`. `t2` is formed by performing a `RIGHT JOIN` on two identical tables with a join condition that always evaluates to `FALSE`. The `ON FALSE` predicate ensures the query result for `t2` contains one row with all values set to `NULL`. Consequently, performing `t1 RIGHT JOIN t2` produces one row where both `t1.c0` and `t2.c0` are `NULL`. In the rewritten query, connecting subquery `q0` to `t1` via a `LEFT JOIN` is semantically equivalent to the `t1 RIGHT JOIN t2` operation in the original query. However, during the projection phase, `t0.c0` is mistakenly aliased as `c0` instead of the correct alias `c1`, introducing a syntax error.

```
-- Misusing the same column name in subquery
CREATE TABLE t0 (c0 smallint DEFAULT NULL);
CREATE TABLE t1 (c0 smallint DEFAULT NULL);
INSERT INTO t0 VALUES (0);
INSERT INTO t1 VALUES (0);
-- Original query
SELECT t2.c0 FROM t1 RIGHT JOIN (
    SELECT t3.c0 AS c0 FROM t0 AS t3
    RIGHT JOIN t0 ON FALSE
    WHERE t0.c0 IS NOT NULL
) AS t2 ON (t1.c0=t2.c0)
WHERE t1.c0 IS NULL;
-- Result: {NULL}
-- Rewritten query
SELECT q0.c0 AS c0 FROM (
    SELECT t0.c0 AS c0, t3.c0 AS c0 FROM t0 AS t0
    LEFT JOIN t0 AS t3 ON FALSE
    WHERE NOT t0.c0 IS NULL
) AS q0 LEFT JOIN t1 AS t1 ON q0.c0=t1.c0
WHERE t1.c0 IS NULL;
-- Result: java.sql.SQLSyntaxErrorException
```

Fig. 5. **Invalid SQL Output in WeTune.** During the rewriting process, the column names of the result set of subquery `q0` are mistakenly all labeled as `c0`. This leads to an exception being triggered when performing the projection operation on `q0.c0`.

*(2) A Semantic Deviation in Apache Calcite.* As shown in Figure 6, a semantic deviation is uncovered in Apache Calcite due to an incorrect query rewrite involving both constant columns and a subquery. The original query constructs a nested structure where the inner subquery `t0` generates a result set with two columns, with `t0.c0` being a constant column. The outer query then projects a new column structure, with a constant column 34 while repurposing the inner `t0.c0` as output column `c1`, producing `{34, 49}`. However, the Apache Calcite erroneously assumes the outer query layer is redundant. Therefore, in the rewritten query, the rewriter

TABLE II
ARG FOUND 38 BUGS FROM FOUR QUERY REWRITERS

| # | Rewriter | Bug Type | Bug Description |
|---|---|---|---|
| 1 | Apache Calcite | Invalid SQL Output | Rewritten query syntax error triggering java.sql.SQLException: Subquery returns more than 1 row. |
| 2 | Apache Calcite | Invalid SQL Output | Invalid column name in rewritten query triggers java.sql.SQLSyntaxErrorException. |
| 3 | Apache Calcite | Invalid SQL Output | Rewritten query with syntax errors like `CAST(NULL AS INTEGER)`. |
| 4 | Apache Calcite | Invalid SQL Output | `SELECT` clause of rewritten query contains non-grouped columns without `GROUP BY` specification. |
| 5 | Apache Calcite | Invalid SQL Output | Rewritten query triggerjava.sql.SQLIntegrityConstraintViolationException since ambiguous column. |
| 6 | Apache Calcite | Semantic Deviation | Projection errors lead to column count or order mismatch between original and rewritten queries. |
| 7 | Apache Calcite | Semantic Deviation | Mishandling boolean logic in `ON` clauses, e.g., erroneously deleting `(ON FALSE)` predicate. |
| 8 | Apache Calcite | Semantic Deviation | Distinct clause inconsistency between original and optimized queries induces semantic deviation. |
| 9 | Apache Calcite | Semantic Deviation | Inconsistent query results occurred due to improper use of data type conversion `(CAST(c0 AS REAL))`. |
| 10 | Apache Calcite | Semantic Deviation | Incorrectly retain only the subquery in nested queries. |
| 11 | Apache Calcite | Semantic Deviation | Rewritten query deletes original's `LIMIT` clause when `SELECT` has only constants. |
| 12 | Apache Calcite | Semantic Deviation | The entire query after `UNION/EXCEPT` is incorrectly deleted. |
| 13 | WeTune | Invalid SQL Output | Causing a java.sql.SQLSyntaxErrorException in DBMS due to duplicate field names in the result set. |
| 14 | WeTune | Invalid SQL Output | Rewritten query syntax error triggering java.sql.SQLException: Subquery returns more than 1 row. |
| 15 | WeTune | Invalid SQL Output | Invalid column name in rewritten query triggers java.sql.SQLSyntaxErrorException. |
| 16 | WeTune | Rewriter Crash | Memory leaks in the ANTLR4 library integrated in WeTune trigger crashes during prolonged execution. |
| 17 | WeTune | Rewriter Crash | Large values in schema (e.g. `INSERT INTO t0 VALUE(9999999999)`) trigger numeric overflow crashes. |
| 18 | WeTune | Semantic Deviation | Unable to correctly rewrite `WHERE` clause boolean combinations like `(false OR false)`. |
| 19 | WeTune | Semantic Deviation | Unable to correctly rewrite `RIGHT JOIN` and `INNER JOIN`, resulting in non-equivalent rewritten queries. |
| 20 | WeTune | Semantic Deviation | Distinct clause inconsistency between original and optimized queries induces semantic deviation. |
| 21 | WeTune | Semantic Deviation | Mistakenly swaps table positions in `RIGHT JOIN` clauses, breaking query equivalence. |
| 22 | WeTune | Semantic Deviation | Incorrect pruning of filter conditions in `WHERE` clauses. |
| 23 | WeTune | Semantic Deviation | Operator priority parsing errors, e.g., mishandling `((c0 > 0) LIKE 0)` as `(c0 > 0 LIKE 0)`. |
| 24 | SQLSolver | Invalid SQL Output | Rewritten query syntax error triggering java.sql.SQLException: Subquery returns more than 1 row. |
| 25 | SQLSolver | Invalid SQL Output | Invalid column name in rewritten query triggers java.sql.SQLSyntaxErrorException. |
| 26 | SQLSolver | Rewriter Crash | Memory leaks in the ANTLR4 library integrated in WeTune trigger crashes during prolonged execution. |
| 27 | SQLSolver | Semantic Deviation | Incorrect pruning of filter conditions in `WHERE` clauses. |
| 28 | SQLSolver | Semantic Deviation | Unable to correctly rewrite `RIGHT JOIN` and `INNER JOIN`, resulting in non-equivalent rewritten queries. |
| 29 | SQLSolver | Semantic Deviation | Mistakenly swaps table positions in `RIGHT JOIN` clauses, breaking query equivalence. |
| 30 | SQLSolver | Semantic Deviation | Distinct clause inconsistency between original and optimized queries induces result mismatch. |
| 31 | SQLSolver | Semantic Deviation | Unable to correctly rewrite `WHERE` clause boolean combinations like `(false OR false)`. |
| 32 | LearnedRewrite | Invalid SQL Output | Causing a java.sql.SQLSyntaxErrorException in DBMS due to duplicate field names in the result set. |
| 33 | LearnedRewrite | Invalid SQL Output | Invalid column name in rewritten query triggers java.sql.SQLSyntaxErrorException. |
| 34 | LearnedRewrite | Invalid SQL Output | Using incorrect structure `VALUES (NULL)` causes the syntax error. |
| 35 | LearnedRewrite | Rewriter Crash | A field ordinal binding failure triggers an AssertionError, crashing the program. |
| 36 | LearnedRewrite | Semantic Deviation | Mishandling boolean logic in `ON`/`WHERE` clauses, e.g., erroneously deleting `(ON FALSE)` predicate. |
| 37 | LearnedRewrite | Semantic Deviation | Non-equivalence caused by the addition of unnecessary joins in the original query. |
| 38 | LearnedRewrite | Semantic Deviation | Inconsistent query results occurred due to improper use of data type conversion `(CAST(c0 AS REAL))`. |

erroneously extracts only the subquery while neglecting the fact that `t0.c1` remains unused in the original query, causing the rewritten query to return a different result set than the original, thus violating query equivalence.

```
-- Incorrectly retain only the subquery
CREATE TABLE t1 (c0 decimal(10, 0), c1 varchar(500));
INSERT INTO t1 VALUES (3415, NULL);
-- Original query
SELECT 34 AS c0, t0.c0 AS c1 FROM (
    SELECT 49 AS c0, t1.c0 AS c1 FROM t1 LIMIT 87
) AS t0;
-- Result: {34, 49}
-- Rewritten query
SELECT 49 AS c0, c0 AS c1 FROM t1 LIMIT 87;
-- Result: {49, 3415}
```

Fig. 6. **Semantic deviation in Apache Calcite.** During rewriting, Apache Calcite incorrectly retains only the original query's subquery, making the statement not equivalent before and after rewriting.

*Why only* ARG *discovered these bugs.* ARG uniquely detects these bugs due to its rewriting-aware design and feedback-driven query generation. Unlike traditional fuzzers that generate queries randomly or rely on code coverage, ARG extracts and uses abstract rewriting rules to guide query generation, systematically activating complex rewriting logic. By incorporating schema constraints and semantic context, it produces realistic queries that expose subtle bugs. Additionally, its semantic validation precisely compares original and rewritten query results, identifying errors missed by syntax-based or coverage-driven methods.

### C. Comparison with Other Techniques

To the best of our knowledge, ARG is the first dedicated tool for testing query rewriters. Therefore, to evaluate the effectiveness of our approach, we compared ARG with two state-of-the-art SQL generators, SQLsmith and SQLancer, both widely used in the industry for generating large volumes of SQL queries that are fed into rewriters for transformation. During the evaluation, all tools were initialized with an empty database as input, executed under their default configurations, and run in the same environment for 24 hours. As the testing proceeds, the data may vary due to randomness and differences in generation strategies. For a fair comparison, after SQLancer

and SQLsmith generated SQL queries, we sent these queries to the target rewriter to collect coverage data and bug counts.

**Coverage.** Table III shows the number of covered branches on four rewrite systems exercised by each tool. ARG significantly outperforms SQLsmith and SQLancer in terms of rule coverage across all four systems. Specifically, ARG covered 18% and 15% more branches in total compared to SQLsmith and SQLancer, respectively. This result demonstrates that ARG explores a broader and deeper range of rewriting logic paths, increasing the likelihood of revealing correctness bugs.

TABLE III
NUMBER OF BRANCHES COVERED BY EACH TOOL IN 24 HOURS

| Rewriter | SQLsmith | SQLancer | ARG |
|---|---|---|---|
| Calcite | 7208 | 7191 | 9703 |
| WeTune | 2617 | 2727 | 2810 |
| SQLSolver | 2869 | 3045 | 3119 |
| LearnedRewrite | 8721 | 8995 | 9661 |
| Total | 21415 | 21958 | 25293 |
| Improvement | 18%↑ | 15%↑ | - |

The primary reason for ARG's improved code coverage is its ability to cover more rewrite rules than other tools. Figure 7 shows the number of unique rewrite rules exercised by each tool. ARG consistently covers more rewrite rules than the other two tools. Specifically, ARG covers a total of 76% and 1017% more abstract rules than SQLsmith and SQLancer, respectively. Furthermore, we also evaluate the coverage of the actual rewriting rules within each rewriter. Table IV shows the number of unique combinations of real rewriting rules exercised by each tool. ARG covers a total of 53% and 476% more unique rewriting rule combinations than SQLsmith and SQLancer, respectively. These results highlight the advantage of ARG's abstract rule-guided generation strategy, which intentionally targets equivalence-preserving transformation logic rather than relying on random or grammar-driven query synthesis.

TABLE IV
NUMBER OF UNIQUE REWRITING RULE COMBINATIONS IN REWRITER BY EACH TOOL IN 24 HOURS

| Rewriter | SQLsmith | SQLancer | ARG |
|---|---|---|---|
| Calcite | 7548 | 1824 | 11544 |
| WeTune | 289 | 44 | 324 |
| SQLSolver | 370 | 56 | 402 |
| LearnedRewrite | 814 | 469 | 1524 |
| Total | 9021 | 2393 | 13794 |
| Improvement | 53%↑ | 476%↑ | - |

**Triggered Bugs.** Table V presents the number of unique rewriter-specific bugs detected by each tool. As shown in the table, ARG detects 13 and 15 more bugs than SQLsmith and SQLancer, respectively. SQLsmith primarily generates standalone `SELECT` statements through random expression synthesis, which limits its ability to cover complex rewrite

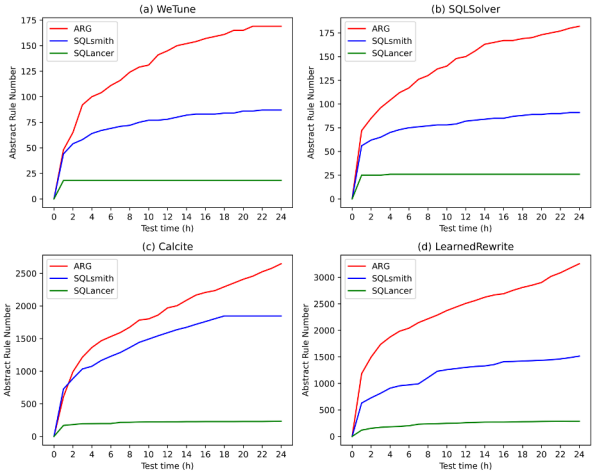


Fig. 7. **Results of Different Tools in Extracting Abstract Rules under a 24-Hour Experiment.** We tracked the growth in the number of unique abstract rules extracted by SQLancer, SQLsmith, and ARG during a 24-hour experiment. ARG demonstrated a significant advantage in abstract rule extraction capability.

rules that often require specific structural patterns or contextual constraints. Similarly, SQLancer focuses on generating SQL queries guided by predefined oracles and grammar constraints, making it difficult to explore the full spectrum of rewrite logic, especially those involving equivalence-preserving transformations. In contrast, ARG systematically identifies and targets rewrite rules through its abstract-rule-guided generation strategy. By analyzing rewriting behavior and validating semantic equivalence between original and rewritten queries, ARG can effectively uncover subtle logic violations, invalid rule applications, and rewriter-internal crashes that are missed by general-purpose SQL generators.

TABLE V
NUMBER OF TRIGGER BUGS BY EACH TOOL IN 24 HOURS

| Rewriter | SQLsmith | SQLancer | ARG |
|---|---|---|---|
| Calcite | 9 | 6 | 12 |
| WeTune | 5 | 8 | 11 |
| SQLSolver | 6 | 4 | 8 |
| LearnedRewrite | 5 | 5 | 7 |
| Total | 25 | 23 | 38 |
| Increment | 13↑ | 15↑ | - |

### D. *Effectiveness of Rule-Guided Strategy*

To assess the effectiveness of the abstract rule guided fuzzing strategy, we implemented a variant of ARG, denoted as ARG-, which disables the rule feedback component. ARG- generates queries using unguided, randomized clause synthesis based solely on grammar structure, without considering rule-triggering feedback or structural transformations observed during rewriting. We compare ARG and ARG- across all four evaluated query rewriters over 24 hours.

Table VI shows the number of rewrite rules exercised by each approach after 24 hours. Across all four rewriters, ARG triggered significantly more rewrite rules than ARG-. Specifically, ARG achieved a 46% improvement in total rule coverage. In addition, ARG achieved a 10% improvement in total code branch coverage compared to ARG-. The main reason behind this improvement lies in ARG's ability to infer and adapt its query generation strategy based on abstract rule feedback. By dynamically analyzing structural transformations between original and rewritten queries, ARG constructs a set of abstract rules that capture equivalence-preserving rewrites. These rules serve as guidance for synthesizing new SQL inputs that are more likely to trigger under-covered or complex rewriting logic. In contrast, ARG- lacks this adaptive mechanism and synthesizes queries randomly, leading to limited exploration of the rewriting space.

TABLE VI
THE NUMBER OF DISTINCT ABSTRACT RULES AND CODE COVERAGE BRANCHES IN ARG AND ARG- OVER 24 HOURS

| Rewriter | Distinct Rules | | Code Coverage | |
|---|---|---|---|---|
| | ARG- | ARG | ARG- | ARG |
| Calcite | 2427 | 2648 | 8053 | 9703 |
| WeTune | 88 | 169 | 2737 | 2810 |
| SQLSolver | 105 | 182 | 3055 | 3119 |
| LearnedRewrite | 1661 | 3256 | 9158 | 9661 |
| Total | 4281 | 6255 | 23003 | 25293 |
| Improvement | 46%↑ | - | 10%↑ | - |

## VI. DISCUSSION

**DBMS Buggy Behavior May Cause Result Mismatch.** We compare original and rewritten query results to detect inconsistencies introduced by rewriting. Given that potential logic bugs in DBMS or their inherent implementation characteristics (e.g., query optimization strategies, incomplete support for specific syntax) might manifest as semantic equivalence deviations, the detected anomalies could stem from underlying database mechanisms rather than the rewriting logic itself. In the evaluation, we detect a total of 39 bugs in the target DBMSs, 38 of which are related to the rewriter, with one bug caused by buggy behavior in MySQL.

Figure 8 illustrates that logically equivalent expressions before and after rewriting can unexpectedly produce different execution results on MySQL. Due to MySQL's implicit type conversion mechanism, the expression `NULLIF(0, 'any text')` evaluates to NULL. These discrepancies stem from underlying buggy behavior in DBMSs rather than bugs in the rewriter implementation. Therefore, when reproducing test results, it is essential to account for database-specific behaviors. We perform cross-database validation by executing rewritten queries across multiple DBMSs, to distinguish rewriter bugs from DBMS-specific behaviors.

**Migration Ability to Other Rewriters.** ARG does not require knowledge of the internal implementation details or rule scheduling mechanisms of query rewriters, making it relatively easy to adapt to other rewriting systems. Specifically, any rewriter that takes an input schema and original query as input and produces one or more rewritten queries as output can be tested by ARG. In our experimental evaluation, we tested both rule-based query rewriters (WeTune, SQLSolver, and LearnedRewrite) and a cost-based query optimizer (Calcite). For rule-based rewriters, given a schema and an input query specification, they deterministically generate rewritten queries based on predefined rewriting rules. In contrast, cost-based rewriters adopt dynamic optimization strategies that consider physical execution metrics such as disk I/O costs, and may produce different rewritten queries even under the same initial conditions. The evaluation results show that ARG is effective in detecting bugs in both types of rewriters, as it does not rely on the specifics of internal scheduling logic but instead focuses on exercising a broad range of rewriting behaviors.

```
CREATE TABLE t0(c0 float);
CREATE TABLE t1(c0 text);
INSERT INTO t0 VALUES (0);
INSERT INTO t1 VALUES ('any text');
-- Original query
SELECT NULLIF(
    c0, COALESCE(c0, (SELECT c0 FROM t1))
) AS c1 FROM t0;
-- Result: {NULL}
-- Rewritten query
SELECT
    CASE WHEN c0 =
        CASE WHEN c0 IS NOT NULL THEN c0
        ELSE ((SELECT * FROM t1)) END
    THEN NULL ELSE c0 END AS c1
FROM t0;
-- Result: {0}
```

Fig. 8. **The Feature in MySQL.** ARG generates a result mismatch due to buggy behavior inherent in MySQL.

## VII. RELATED WORK

**DBMS Fuzzing.** DBMS fuzzing employs two main approaches for query generation: *mutation-based* and *generation-based methods*. Traditional mutation-based tools like AFL [30] often generate semantically invalid SQL that DBMSs or rewriters cannot parse or fully test. To enhance the quality of query generation, recent research efforts have focused on improving the syntactic validity of mutated SQL queries. SQUIRREL [31] builds an intermediate representation (IR) based on SQL grammar rules and creates syntax-valid SQL queries using mutation techniques from AFL's core engine with coverage feedback, effectively exposing crashes in DBMS. SQLRight [13] employs syntax tree mutation for test case generation and adapts inputs based on coverage feedback. GRIFFIN [8] introduces a grammar-free mutation method to generate queries. Generation-based fuzzers generate queries using predefined templates or models. For example, SQLsmith [23] constructs large numbers of `SELECT` statements by populating abstract syntax trees (ASTs) with metadata to test DBMSs. SQLancer [22] leverages predefined syntax tree models to produce grammatically valid SQL

queries, while also creating schemas and populating the database with data. APOLLO [11] focuses more on generating complex or adversarial SQL structures that are likely to degrade the performance of the DBMS.

While existing approaches are effective, they face limitations when applied to standalone query rewriters. Neither mutation-based nor generation-based methods can sufficiently activate internal rewrite rules or thoroughly test the complex behaviors of rewriters. In contrast, ARG guides test case generation through abstract rules, exploring more rewrite logic within the rewriter.

**Differential Testing.** Differential testing validates DBMS by executing the same test case in different implementations to verify if they yield matching results [6], [16]. Any unexpected output reveals a potential bug in the system. RAGS [20] detects system bugs by executing identical queries across different DBMS instances. Gu et al. evaluate query optimizer quality through comparative analysis of execution plans under varied hint configurations [9]. APOLLO [11] identifies performance issues by comparing the execution durations of identical SQL queries across multiple versions of the target DBMS.

Differential testing is difficult to directly apply to query rewriters. Substantial rewriting logic differences cause the same query to yield diverse outputs, rendering traditional differential testing ineffective. ARG focuses on checking the semantic equivalence between the original and rewritten query, which shares similarity with differential testing.

**Metamorphic Testing.** Recent works have also adopted metamorphic testing to construct semantically equivalent queries to test DBMSs. NoREC [18] converts an optimizable query into a non-optimizable form of the query, then identifies logic bugs by comparing their execution results. Similarly, TLP [19] employs ternary logic to generate three equivalent queries combined via UNION operations. Amoeba [14] targets DBMS performance anomalies by crafting logically equivalent queries and examining variations in their runtime behavior TQS [25] detects logical bugs by generating semantically equivalent multi-table join query pairs and checking for inconsistent execution results. GRev [15] automatically tests graph databases by rewriting queries into logically equivalent forms based on a unified graph abstraction, successfully uncovering multiple previously unknown vulnerabilities.

ARG differs from these tools by treating the rewritten query as output and directly verifying semantic equivalence, instead of using predefined metamorphic relations to generate inputs. Therefore, ARG focuses on detecting semantic inconsistencies in query rewriting, rather than testing general input-output transformations as in metamorphic testing.

## VIII. CONCLUSION

Query rewriters frequently contain subtle bugs, but existing testing techniques are insufficient, leaving many bugs undetected and potentially causing incorrect query results. To address this, we propose ARG, the first novel fuzzing framework that automatically extracts abstract rewrite rules by comparing query syntax trees and leverages feedback-driven test generation to maximize rule coverage without requiring knowledge of the rewriter's internal logic. Our evaluation shows that ARG effectively detects 38 previously unknown bugs across multiple real-world query rewriters, outperforming state-of-the-art testing tools. ARG is both intuitive and highly extensible, allowing straightforward adaptation to test arbitrary query rewriters. In the future, we plan to extend ARG to further test SQL optimizers, with the goal of identifying semantic errors and suboptimal plan generation.

## ACKNOWLEDGMENT

This work was supported in part by the National Key Research and Development Program of China under Grant 2021YFB2700200, in part by the Natural Science Foundation of China under Grants T2425023, 62302256, L251040, U2241213, U21B2021, 62372022 and 62172025.

## REFERENCES

[1] Mysql query rewrite plugin. https://dev.mysql.com/doc/refman/8.4/en/rewriter-query-rewrite-plugin.html, 2025.

[2] APACHE. Calcite. https://github.com/apache/calcite.git, 2025.

[3] BEGOLI, E., CAMACHO-RODRÍGUEZ, J., HYDE, J., MIOR, M. J., AND LEMIRE, D. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 221–230.

[4] CALVANESE, D., DE GIACOMO, G., LENZERINI, M., AND VARDI, M. Y. What is query rewriting? In *International Workshop on Cooperative Information Agents* (2000), Springer, pp. 51–59.

[5] DING, H., WANG, Z., YANG, Y., ZHANG, D., XU, Z., CHEN, H., PISKAC, R., AND LI, J. Proving query equivalence using linear integer arithmetic. *Proceedings of the ACM on Management of Data 1*, 4 (2023), 1–26.

[6] EVANS, R. B., AND SAVOIA, A. Differential testing: a new approach to change detection. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers* (New York, NY, USA, 2007), ESEC-FSE companion '07, Association for Computing Machinery, p. 549–552.

[7] FINANCE, B., AND GARDARIN, G. A rule-based query rewriter in an extensible dbms. In *Proceedings. Seventh International Conference on Data Engineering* (1991), IEEE Computer Society, pp. 248–249.

[8] FU, J., LIANG, J., WU, Z., WANG, M., AND JIANG, Y. Griffin: Grammar-free dbms fuzzing. In *Conference on Automated Software Engineering (ASE'22)* (2022).

[9] GU, Z., SOLIMAN, M. A., AND WAAS, F. M. Testing the accuracy of query optimizers. In *Proceedings of the Fifth International Workshop on Testing Database Systems* (2012), pp. 1–6.

[10] IBM. Db2 for linux, unix and windows. https://www.ibm.com/docs/en/db2/11.5.x?topic=process-query-rewriting-methods-examples, 2025.

[11] JUNG, J., HU, H., ARULRAJ, J., KIM, T., AND KANG, W. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems (to appear). In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)* (Tokyo, Japan, Aug. 2020).

[12] LEARNEDREWRITE. https://github.com/XuanheZhou/LearnedRewrite.git, 2022.

[13] LIANG, Y., LIU, S., AND HU, H. Detecting logical bugs of {DBMS} with coverage-based guidance. In *31st USENIX Security Symposium (USENIX Security 22)* (2022), pp. 4309–4326.

[14] LIU, X., ZHOU, Q., ARULRAJ, J., AND ORSO, A. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 225–236.

[15] MANG, Q., FANG, A., YU, B., CHEN, H., AND HE, P. Testing graph database systems via equivalent query rewriting. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024), ICSE '24, Association for Computing Machinery.

[16] MCKEEMAN, W. M. Differential testing for software. *Digital Technical Journal 10*, 1 (1998), 100–107.

[17] NEGRI, M., PELAGATTI, G., AND SBATTELLA, L. Formal semantics of sql queries. *ACM Trans. Database Syst. 16*, 3 (Sept. 1991), 513–534.

[18] RIGGER, M., AND SU, Z. 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* (2020), pp. 1140–1152.

[19] RIGGER, M., AND SU, Z. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang. 4*, OOPSLA (Nov. 2020).

[20] SLUTZ, D. R. Massive stochastic testing of sql. In *VLDB* (1998), vol. 98, Citeseer, pp. 618–622.

[21] SQL FORMATTER. sql-formatter. https://github.com/vertical-blank/sql-formatter.git, 2024.

[22] SQLANCER. https://github.com/sqlancer/sqlancer.git, 2025.

[23] SQLSMITH. https://github.com/anse1/sqlsmith.git, 2018.

[24] SQLSOLVER. https://github.com/SJTU-IPADS/SQLSolver.git, 2022.

[25] TANG, X., WU, S., ZHANG, D., LI, F., AND CHEN, G. Detecting logic bugs of join optimizations in dbms. *Proceedings of the ACM on Management of Data 1*, 1 (2023), 1–26.

[26] WANG, Z., ZHOU, Z., YANG, Y., DING, H., HU, G., DING, D., TANG, C., CHEN, H., AND LI, J. Wetune: Automatic discovery and verification of query rewrite rules. In *Proceedings of the 2022 International Conference on Management of Data* (2022), pp. 94–107.

[27] WETUNE. https://ipads.se.sjtu.edu.cn:1312/opensource/wetune.git, 2022.

[28] WIKIPEDIA. Query rewriting. https://en.wikipedia.org/wiki/Query_rewriting, 2025.

[29] XIAOMI. Soar: Sql optimizer and rewriter. https://github.com/xiaomi/soar, 2025.

[30] ZALEWSKI, M. Afl: American fuzzy lop. https://github.com/google/AFL, 2015.

[31] ZHONG, R., CHEN, Y., HU, H., ZHANG, H., LEE, W., AND WU, D. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (2020), pp. 955–970.

[32] ZHOU, X., JIN, L., SUN, J., ZHAO, X., YU, X., FENG, J., LI, S., WANG, T., LI, K., AND LIU, L. Dbmind: A self-driving platform in opengauss. *Proceedings of the VLDB Endowment 14*, 12 (2021), 2743–2746.

[33] ZHOU, X., LI, G., CHAI, C., AND FENG, J. A learned query rewrite system using monte carlo tree search. *Proc. VLDB Endow. 15*, 1 (2021), 46–58.

[34] ZHOU, X., LI, G., WU, J., LIU, J., SUN, Z., AND ZHANG, X. A learned query rewrite system. *Proc. VLDB Endow.* (2023).