OraclePlate: Detecting Logic Bugs in DBMSs via Cascaded Rewriting-Based Test Oracle Composition

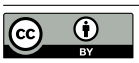
YUXIAO GUO*, Beihang University, China
DAWEI LI*, Beihang University, China
QIFAN LIU, Beihang University, China
JIE LIANG†, Beihang University, China
ZHIYONG WU, Tsinghua University, China
CHI ZHANG, Tsinghua University, China
JINGZHOU FU, Tsinghua University, China
JINHUI LAI, Nanchang University, China
WENQIAN DENG, Tsinghua University, China
YU JIANG, Tsinghua University, China
ZHENYU GUAN, Beihang University, China

Logic bugs in DBMSs are subtle implementation errors that silently produce incorrect results, posing severe threats to upper-layer enterprise applications. Unlike crashes, these bugs lack explicit failure symptoms, making them particularly difficult to detect. Consequently, a core challenge in DBMS testing is the test oracle problem, i.e., how to determine the correctness of query results. Existing approaches mainly construct test oracles based on equivalent relations and detect bugs by comparing the results of equivalent query pairs. However, such approaches are typically restricted to a single equivalence relation and its associated input forms, leaving many bugs hidden in the alternative transformation paths unexplored.

In this paper, we propose OraclePlate, a framework for finding logic bugs in DBMSs via cascaded rewriting-based test oracle composition. The key insight is that diverse equivalence-based test oracles can be uniformly modeled as stochastic-augmented rewriting rules over queries, enabling transformation at valid positions and supporting their cascaded compositions. Specifically, OraclePlate proceeds in three steps. First, it introduces a unified specification *UES* that standardizes diverse test oracles with stochastic rule extensions and enables their composition. Second, it employs a topology-based SQL lineage tracking mechanism to preserve semantic correctness during chained transformations. Finally, it adopts a rule-guided strategy to prioritize high-value transformation paths and mitigate path explosion. We implement OraclePlate on six widely-used and well-tested DBMSs, namely PostgreSQL, DuckDB, MySQL, MariaDB, TiDB and Percona. OraclePlate uncovered 39 previously unknown bugs, all of which have been confirmed. Among them, 35

*Both authors contributed equally to this research.
†Jie Liang is the corresponding author.

Authors' Contact Information: Yuxiao Guo, yuxiaoguo@buaa.edu.cn, Beihang University, Beijing, China; Dawei Li, lidawei@buaa.edu.cn, Beihang University, Beijing, China; Qifan Liu, imchifan@163.com, Beihang University, Beijing, China; Jie Liang, liangjie.mailbox.cn@gmail.com, Beihang University, Beijing, China; Zhiyong Wu, 253540651@qq.com, Tsinghua University, Beijing, China; Chi Zhang, chi-zhang@mail.tsinghua.edu.cn, Tsinghua University, Beijing, China; Jingzhou Fu, fuboat@outlook.com, Tsinghua University, Beijing, China; Jinhui Lai, jinhuilai@email.ncu.edu.cn, Nanchang University, Nanchang, China; Wenqian Deng, dengwenking@gmail.com, Tsinghua University, Beijing, China; Yu Jiang, jiangyu198964@126.com, Tsinghua University, Beijing, China; Zhenyu Guan, guanzhenyu@buaa.edu.cn, Beihang University, Beijing, China.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2836-6573/2026/12-ART365
https://doi.org/10.1145/3856360

bugs are revealed that cannot be exposed by single-oracle approaches, demonstrating the effectiveness of test oracle composition.

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

Additional Key Words and Phrases: DBMS Testing, Test Oracles, Logic Bugs

ACM Reference Format:
Yuxiao Guo, Dawei Li, Qifan Liu, Jie Liang, Zhiyong Wu, Chi Zhang, Jingzhou Fu, Jinhui Lai, Wenqian Deng, Yu Jiang, and Zhenyu Guan. 2026. OraclePlate: Detecting Logic Bugs in DBMSs via Cascaded Rewriting-Based Test Oracle Composition. *Proc. ACM Manag. Data* 4, 6 (SIGMOD), Article 365 (December 2026), 26 pages. https://doi.org/10.1145/3856360

1 Introduction

Database Management Systems (DBMSs) are the backbone of modern data-intensive applications [2, 11, 26]. As data scale and workload complexity continue to grow, DBMS implementations have become increasingly complex, introducing bugs that pose potential reliability risks. Among them, logic bugs are particularly critical. Unlike crashes, logic bugs are hard to detect as they silently produce incorrect results without triggering explicit error signals [30]. Such incorrect results can have severe consequences in applications that require high data accuracy, such as financial transaction systems. Therefore, effectively detecting logic bugs is important for DBMSs.

To detect these bugs, DBMS testing requires *test oracles*, which determine whether query results are correct [5]. Recently, many works have focused on constructing test oracles for DBMSs by exploiting semantic relationships between queries, with equivalence-based methods being the most widely adopted. Semantically equivalent queries may exercise different optimization or execution paths but should produce consistent results. Any discrepancy indicates a potential logic bug. These equivalence relations are typically modeled as metamorphic relations and applied in metamorphic testing [16, 17, 27–29]. In this paradigm, the original query is transformed into semantically equivalent queries using equivalence transformation rules, and potential errors are detected by comparing their execution results. For example, TLP [28] partitions the original query predicate into `True`, `False`, and `IS NULL` branches, and EET [16] introduces redundant branch structures into query expressions in a semantics-preserving manner. Based on these methods, many logic bugs in real-world DBMSs are detected.

However, such approaches are typically restricted to a single equivalent transformation and its associated input forms, leaving many bugs hidden in the alternative rewriting paths unexplored. Specifically, the expressiveness of a single equivalence transformation is often confined to a particular SQL semantic fragment and local operands, whereas logic bugs may only be triggered under specific rewriting sequences or rule interactions. More fundamentally, existing test oracles are standalone procedures whose applicability conditions and symbol dependencies remain embedded in oracle-specific implementations. Without a common representation, one oracle's output lacks the information needed to determine whether another can be applied soundly. Consequently, newly exposed rewrite sites cannot be systematically validated, leaving their composition space unexplored. Even if a DBMS performs correctly with respect to each individual rule, its overall query processing pipeline may still introduce semantic deviations due to the cumulative effects of multi-step transformations. Consequently, such logic bugs cannot be exposed by testing individual equivalence rules in isolation. For example, Figure 1 illustrates a logic bug in MariaDB. Even when applying the state-of-the-art testing oracle EET to support expression-level transformations, using this oracle alone on the original query yields only equivalent results. However, by first applying EET rewriting in the FROM clause and then performing a second EET rewriting in the WHERE clause, we ultimately uncover a new logic bug.

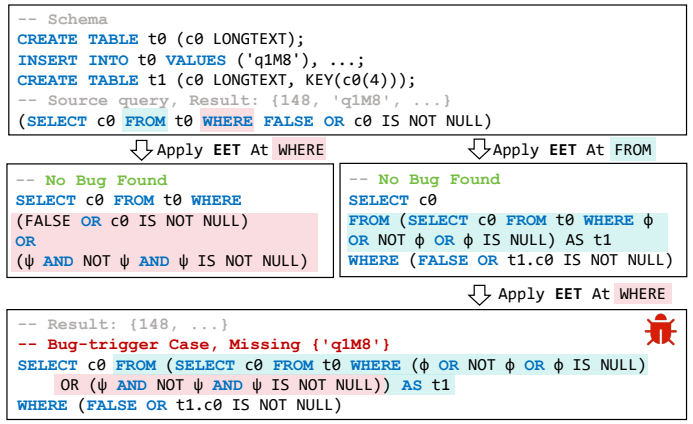


Fig. 1. A logic bug in MariaDB that cannot be found with a single EET oracle, but it can be revealed through a chained composition of test oracles: EET → EET.

In this paper, we propose a compositional approach to construct DBMS test oracles. *The key intuition is that composing multiple test oracles can lead DBMSs to exercise code logic different from that of single oracles, and may expose bugs hidden in these paths.* Realizing the composition has three main challenges. First, unifying structurally diverse test oracles. It is necessary to represent different test oracles to enable composition. But it is difficult because existing oracles rely on heterogeneous SQL structures and semantics, leaving few natural composition sites. Second, ensuring SQL syntactic and semantic validity during chained transformations. Different oracles use diverse transformation forms and granularity that may conflict with each other. Subtle interactions among transformation rules can also cause failures to preserve equivalence. Third, mitigating path explosion during composition. The space of possible transformation sequences grows exponentially when composing multiple oracles. Exhaustive exploration of all paths is inefficient, while arbitrary selection may miss bug-triggering states.

To address these challenges, we propose OraclePlate, a testing framework for finding logic bugs in DBMSs via cascaded rewriting-based test oracle composition. **The key insight is that diverse equivalence-based test oracles can be uniformly modeled as stochastic-augmented rewriting rules, enabling cascaded transformation at valid positions.** Specifically, OraclePlate comprises three key steps: (1) First, it abstracts diverse equivalence-based test oracles into rewriting rules using a *Unified Equivalence Specification (UES)* with stochastic rule extensions. This allows each oracle to be concisely defined by a single-line string, shielding implementation differences and providing a unified form for oracle composition. (2) Second, OraclePlate tracks column and table lineage relationships to ensure reference correctness during transformations. By analyzing query data dependencies and applying topological sorting, OraclePlate extracts rewriting metadata and thereby strictly maintains semantic validity. (3) Finally, OraclePlate dynamically tracks the occurrence of unique rule paths to guide the chained rewriting process. By continuously exploring new rule combinations, OraclePlate effectively mitigates the path explosion problem and targets deep paths to expose logic bugs.

We evaluate OraclePlate on six widely-used and well-tested DBMSs: PostgreSQL, DuckDB, MySQL, MariaDB, TiDB and Percona. OraclePlate detected 39 previously unknown bugs, including 35 logic bugs and 4 crashes. All 39 bugs have been confirmed by developers, and 9 have already been fixed. Among them, 35 bugs are revealed that cannot be exposed by single-oracle approaches, demonstrating the effectiveness of compositional test oracles. We also compare OraclePlate with

state-of-the-art DBMS logic bug testing tools, including TLP, EET, PQS, and SRS. Experimental results show that OraclePlate discovers 5.2%, 22.5%, 14.5%, 5.6% more branches and 25, 28, 28, 20 more bugs than these baselines in 24 hours. Furthermore, ablation experiments demonstrated the effectiveness of the chained rewriting strategy and rule guidance. In summary, we make the following contributions:

- First, we find that many DBMS logic bugs can only be triggered by the compositional effects of multiple equivalence transformations, whereas most existing test oracle methods consider only a single transformation in isolation.
- We propose OraclePlate, a testing framework to find logic bugs via rewriting-based test oracle composition. It unifies test oracles based on *UES* rules, rewrites queries in a cascaded manner with lineage tracking, and explores compositional space with a rule-guided strategy.
- We uncovered 39 unique bugs in six popular DBMSs, with all confirmed by developers and 9 already fixed, demonstrating the practical value of OraclePlate.

2 Background and Motivation

Logic Bugs and Test Oracles in DBMSs. *Logic Bugs* in DBMSs are implementation flaws that cause incorrect query results [30]. Unlike crash bugs, they often lack observable symptoms, making crash-based fuzzing ineffective for their detection. A *test oracle* determines whether a query result is correct [5], typically by comparing the output with an expected result or checking predefined correctness properties. However, constructing precise oracles is often difficult for complex queries, as the expected results may be unknown or expensive to compute.

To address this, prior work has proposed a variety of test oracles for DBMS testing. Early efforts apply differential testing [33], in which the same query is executed across multiple DBMSs to identify inconsistencies. However, this approach is constrained by differences in SQL dialects and system behaviors, making it difficult to identify comparable systems. To overcome the dependency on multiple DBMSs, some methods like PQS [29] construct test oracles by ensuring that query results satisfy semantic invariants. But these approaches rely on rigid predicate construction, which limits their ability to express complex query semantics.

Recently, more work has focused on constructing test oracles based on the semantic relationships among queries in metamorphic testing, especially equivalence relationships. Specifically, if an original query is transformed into a semantically equivalent variant, both should return the same result on the same DBMS state, and any discrepancy strongly suggests a logic bug. For example, TLP (Ternary Logic Partitioning) partitions query results using TRUE, FALSE, and `IS NULL` [28]. SRS (Set Relation Synthesis) detects bugs by checking set relations among transformed join queries [17]. EET (Equivalence Expression Transformation) constructs semantically equivalent queries by rewriting expressions into alternative but equivalent forms [16].

Limitations of Existing Test Oracles. Although existing test oracles have detected many bugs, they are typically designed and applied as standalone transformations. As a result, they are restricted to a single equivalence rule and its associated input forms, leaving many bugs hidden in the alternative rewriting paths unexplored.

Taking Figure 2 as an illustrative example, it presents a query that triggers a logic bug hidden in the optimizer of MySQL. Although the query shown in the figure has been simplified, it still exhibits a fairly complex structure, involving nested subqueries, set operations (e.g., `UNION ALL`), and correlated predicates (e.g., `EXISTS`). The original query is relatively simple. After being transformed sequentially using rules from EET, TLP, and SRS (applied twice), it is transformed into a more complex form. This composed transformation chain introduces non-trivial structural

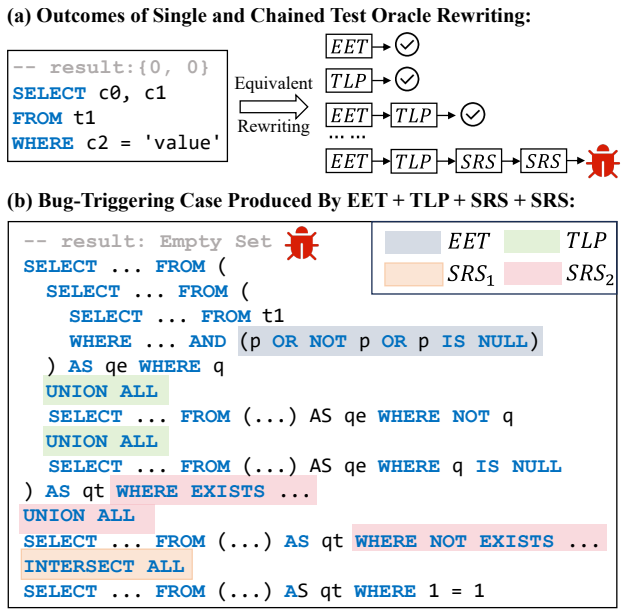


Fig. 2. The case that triggered a logic bug in MySQL. It is exposed by a composed test oracle chain EET → TLP → SRS → SRS generated by OraclePlate.

variations, including predicate partitioning (from TLP), join and set-relation rewritings (from SRS), and expression-level equivalences (from EET), ultimately exercising a faulty execution path.

Notably, none of these oracles alone is sufficient to trigger the bug. EET is limited to local expression rewrites (e.g., expanding `p` to `p AND (q OR NOT q OR q IS NULL)`) and does not substantially alter the global query structure. TLP partitions predicates (e.g., into `q`, `NOT q`, and `q IS NULL`) but lacks the structural diversification needed to expose faulty interactions with other operators. SRS generates set-preserving join variants but cannot produce the specific combination of nested subqueries and set operators required to trigger the bug. Only when these transformations are composed, gradually increasing both structural complexity and semantic coverage, does the resulting query reach a form that exposes the latent defect. As a result, exploring composed transformation paths is an intuitive direction to improve logic bug finding capabilities.

Challenge For Oracle Composition. Composing different test oracles is non-trivial and mainly faces three challenges. *(1) First, unifying structurally diverse test oracles.* To enable composition, it is necessary to represent different test oracles within a unified formalism. However, existing test oracles vary substantially across approaches and are often highly system-dependent. As shown in Table 1, adapting a single oracle to a DBMS often requires thousands of lines of code. Due to the substantial implementation effort involved, many test oracles remain unavailable for popular DBMSs. (2) *Second, ensuring SQL syntactic and semantic validity during chained transformations.* Different test oracles employ diverse transformation forms and rewriting granularities, which may introduce conflicts when combined. During chained rewriting, it is highly challenging to ensure that SQL statements generated after multiple transformations remain valid without disrupting the original query intent. (3) *Third, mitigating path explosion in transformation.* As the number of participating oracles increases, the number of possible rewriting paths grows exponentially. Exploring all transformation sequences without restriction leads to path explosion and substantially reduces testing efficiency. In contrast, selecting sequences arbitrarily may fail to reach meaningful

intermediate states that expose bugs. Therefore, it is necessary to design reasonable scheduling strategies to ensure test coverage while avoiding ineffective transformation spaces.

Table 1. The lines of code to implement and adapt different test oracles to various DBMSs.

Test Oracle	PostgreSQL	DuckDB	MySQL	MariaDB	TiDB	Percona
TLP [28]	531	541	624	Unsupported	743	Unsupported
PQS [29]	289	Unsupported	2102	Unsupported	Unsupported	Unsupported
EET [16]	5576	Unsupported	5288	Unsupported	5030	Unsupported
SRS [17]	617	Unsupported	877	620	651	Unsupported

Solution of OraclePlate. The key insight of OraclePlate is that *equivalence-based test oracles can be uniformly modeled as query rewriting rules with stochastic generation.* This representation allows rules derived from different test oracles to be formalized in a common form and successively applied to the same query, enabling cascaded transformations.

Specifically, to address challenge 1, OraclePlate unifies diverse oracle implementations into a unified rewriting form *UES*, so that the rewriting output of one oracle can be directly consumed by the next. For challenge 2, OraclePlate tracks both column and table lineage to ensure reference correctness and uses dependency analysis with topological sorting to generate rewriting metadata, thereby preserving semantic validity during transformations. To address challenge 3, OraclePlate dynamically tracks the occurrence frequency of unique rule paths to guide the subsequent chained rewriting process.

Figure 2 illustrates this process, with different colors denoting structures introduced by different test oracle transformations. For example, starting from the query "`SELECT c0, c1 FROM t1 WHERE c2 = 'value'`", OraclePlate identifies a position in the `WHERE` predicate to which EET or TLP can be applied. The initial rewrite produces no result mismatch, so OraclePlate continues rewriting the transformed queries. Subsequent transformations expose additional nested sites for TLP, EET, and SRS, progressively producing equivalent queries with nested subqueries, `UNION ALL`, `INTERSECT ALL`, and predicates such as "p OR NOT p OR p IS NULL". After the third and fourth SRS-based rewrites, OraclePlate eventually detects a non-equivalent result and finds the bug.

3 Design of OraclePlate

Figure 3 illustrates the overall process of OraclePlate, which contains three steps: 1) rewriting-driven test oracle unification, 2) lineage-aware query rewriting, and 3) path-guided cascaded oracle composition. The following text presents the details.

3.1 Rewriting-Driven Test Oracle Unification

Existing test oracles are typically implemented as isolated, system-dependent code tightly coupled to specific transformation strategies and DBMS dialects, and are difficult to compose or to support further query transformations to explore deeper execution paths in DBMSs. Query rewriting rules are designed to express equivalence-preserving transformations. Consequently, it is intuitive to consider whether they can serve as a unified foundation for representing and composing diverse equivalence-based test oracles.

However, existing query rewriters [6, 15, 37] fall short of this goal. Specifically, traditional deterministic rewriters are closed over existing query symbols and mainly perform local algebraic substitutions, so they cannot express scoped symbol generation, cross-branch binding, and constraint-aware instantiation in a unified representation. For example, Figure 4 illustrates the gap

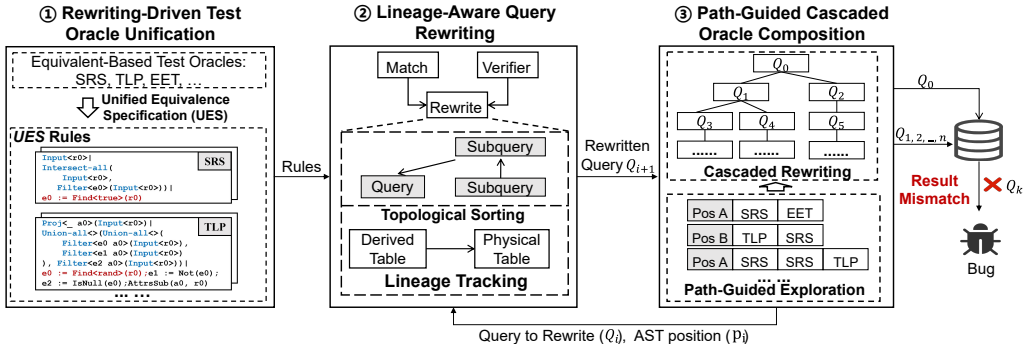


Fig. 3. Overview of OraclePlate. OraclePlate consists of three steps to compose test oracles: (1) To compose structurally diverse test oracles, OraclePlate introduces *UES* with stochasticity rule extensions to unify them into a common form. (2) To preserve correctness under chained transformations, OraclePlate performs lineage-aware query rewriting with dependency tracking to ensure both semantic validity and equivalence. (3) To mitigate path explosion in oracle composition, OraclePlate adopts a path-guided scheduling strategy to explore new rewriting paths that are hard for standalone oracles to reach. Any inconsistency between rewritten and original query results is then treated as a potential logic bug.

in existing SQL rewriters to represent the TLP semantics. Figure 4 (a) shows that TLP is not merely a structural rewrite from one query into three UNION branches: it must first synthesize a fresh predicate symbol e_0 (e.g., $t0.c0 > 10$), then propagate it consistently as e_0, $\neg e_0$, and e_0 `IS NULL`. Figure 4(b) further shows the relational algebra view of this limitation. In existing rewriters, instantiation is typically performed by tracing each symbol in the rewritten expression back to a corresponding symbol in the source query. Under this mechanism, the R_0 in the rewritten algebra is not created independently, but instantiated from the original R_0 in the input query through such backward symbol resolution. However, the predicate symbol e_0 would need to be instantiated in the same way. Since no corresponding predicate symbol exists in the original query, the rewriter cannot trace e_0 back to any source symbol or expression. This limitation motivates our **Unified Equivalence Specification (*UES*)** design with stochastic generation via the `Find` operator. *UES* abstracts diverse test oracles into a unified representation, providing a standardized protocol for test oracle composition.

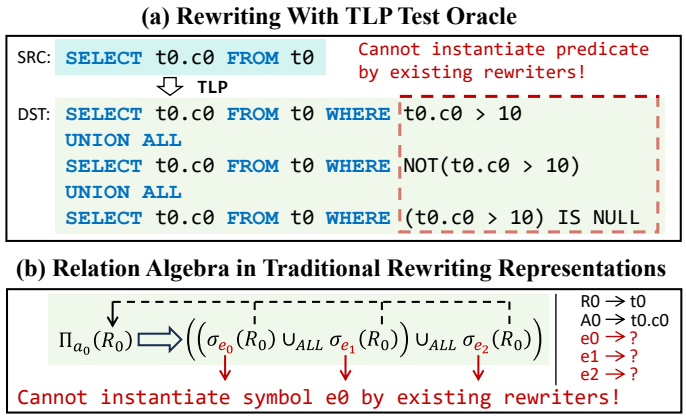


Fig. 4. Existing rewriters can't instantiate predicates or bind symbolic variables in representing TLP semantics.

Table 2. Core elements of the *UES*.

Element	Specification and semantics
Template	`Input`, `Filter`, `Proj`, `Join`, `Agg`, `Union`, and `SubQuery` construct the source and destination AST patterns. They represent relational operations such as R, $\sigma_e(R)$, $\Pi_a(R)$, $R_l \bowtie_e R_r$, ${}_g\gamma_f(R)$, and $R_l \cup R_r$.
Symbol	r_i, a_i, and e_i denote typed placeholders for relations, attributes, and expressions, respectively.
Find	$\boldsymbol{e :=}$ **Find**$\boldsymbol{\langle cat \rangle(r)}$ generates a well-typed AST fragment of category *cat* from the schema, scope, and visible columns of relation r.
Constraint	$[\neg]\mathsf{Type}(s_1, s_2, \ldots)$ checks semantic conditions such as attribute containment, equality, uniqueness, and nullability.
Binding	$s := \mathsf{Exp}(\ldots)$ constructs expressions, while $\mathsf{Exp}(\ldots) := s$ decomposes matches and binds symbols.

3.1.1 Design Principle for Oracle Composition. The design principle of *UES* is to abstract oracle-specific semantics into a unified rewrite interface. A future oracle can be composed in *UES* if it satisfies three requirements. (1) An oracle must define a bug-detection criterion expressible as result-set equivalence. (2) The oracle must provide an equivalence-preserving rewrite that captures the intended semantics. (3) All applicability conditions must be checkable without relying on runtime behavior, such as transactions or nondeterministic execution.

3.1.2 Unified Equivalence Specification. In our design, test oracles are no longer hard-coded transformation logic, but rather declarative *UES* rules. We define a *UES* rule as a tuple

$$r = \langle T_{src}, T_{dst}, C \rangle$$

where T_{src} is the source template, T_{dst} is the target template, and C is a set of explicitly specified rule clauses. Both T_{src} and T_{dst} are built from relational-algebra-style operators (e.g., `Filter`, `Proj`, `Union`, and `Join`), which provides a uniform structural basis for semantics-preserving rewriting. The clause set C contains three parts: (1) *guard constraints* that determine whether a rule instance is valid, (2) *binding clauses* that bind symbols by decomposing matched expressions and constructing derived expressions, and (3) *generation directives* (e.g., `Find`) for schema-aware and type-aware nondeterministic instantiation.

Formally, for a query q, the rule is applicable if q matches T_{src} under a substitution θ and all clauses in C are satisfied; the rewritten query is then obtained by instantiating T_{dst} with θ and generated symbols. This design decouples oracle definition from engine code and allows heterogeneous test oracles to be represented and composed through one unified interface.

AST-Style Representation and Rule Elements. As shown in Figure 5, *UES* uses a normalized AST representation with relational-algebra-style operators because OraclePlate must regenerate executable SQL across dialects while preserving SQL-specific properties, such as column visibility and `ALL`/`DISTINCT` semantics. These properties are not uniformly represented in relational algebra, while internal logical plans are DBMS-specific and unsuitable as a portable basis for SQL regeneration. As summarized in Table 2, T_{src} and T_{dst} combine normalized AST operators with typed relation, attribute, and expression placeholders, while C specifies applicability constraints, symbol bindings, and generation directives such as `Find`. Unlike SQL templates, this representation makes symbol types, cross-branch bindings, scope constraints, and stochastic generation explicit, enabling validation and reuse in subsequent rewrites.

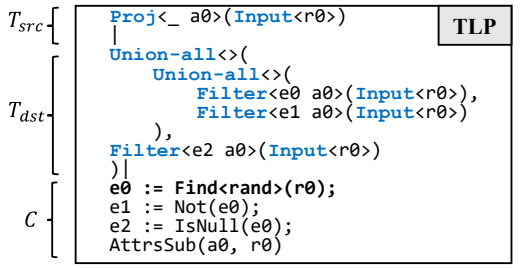


Fig. 5. The *UES* representation of the test oracle TLP.

Stochastic Generation via the `Find` Operator. Unlike traditional static query rewriting rules (e.g., query optimization rules in GRewriter [15] or WeTune [37]) that perform deterministic algebraic transformations, test oracles inherently rely on stochastic mechanisms to expand the testing space and trigger logic bugs. For example, TLP [28] requires synthesizing a random boolean predicate to partition the result set into mutually exclusive branches.

To support this requirement, we introduce the `Find` operator into *UES*. The `Find` operator applies a controlled nondeterministic instantiation step into *UES* rules. During rule instantiation, it invokes the generator to produce well-typed Abstract Syntax Tree (AST) fragments that satisfy the requested syntactic category and all currently bound scope information and type constraints. Specifically, the operator `e := Find<cat>(r)` accepts a target structural category (e.g., `predicates`, `expressions`) and a relational context r. By strictly analyzing the schema of r and the real data in the physical tables corresponding to r in the database, the underlying generator guarantees that the synthesized AST fragment only references available columns and adheres to correct type constraints, effectively preventing semantic invalidation (e.g., "column not found" errors) during random generation.

For each candidate query, OraclePlate performs AST-level rule instantiation to construct valid query fragments in three steps. 1) *AST matching.* It traverses the AST to identify subtrees matching a *UES* rule's source template T_{src} and satisfying the constraints in C. 2) *Fragment generation.* For unbound symbols (e.g., predicate e), `Find` synthesizes a well-typed AST fragment under the appropriate relation context r. 3) *AST integration.* The generated fragments are substituted into the matched subtree, after which OraclePlate synchronizes LID-based column lineage and scope information and regenerates the final SQL statement. Further details on lineage synchronization are presented in Section 3.2.

Figure 5 shows the *UES* representation of TLP, consisting of a source pattern T_{src}, a destination pattern T_{dst}, and a constraint set C. In this case, T_{src} matches queries of the form `Proj<_` a_0`>(Input<`r_0`>)`, where r_0 denotes the input relation context and a_0 denotes the projected attributes. T_{dst} rewrites the matched query into three `Filter` branches over the same `Input<`r_0`>` and combines them using two nested `Union_all` nodes, illustrating how *UES* represents structures with more than two inputs through recursive binary composition. In this way, the rule preserves the original query interface while capturing the core TLP three-branch structure. The constraint set C then determines how these symbols are instantiated. In particular, e_0 `:= Find<rand>(`r_0`)` enables TLP by synthesizing a fresh well-typed boolean predicate under the relation context r_0, rather than tracing e_0 back to the source query. Once generated, this predicate is consistently reused across the rewrite, and `e1 := Not(`e_0`)` and e_2 `:= IsNull(`e_0`)` derive the other two branches required by SQL three-valued logic. Finally, `AttrsSub(`a_0`,` r_0`)` ensures that all projected attributes remain valid under r_0. Together, these constraints make the rule executable while preserving the core TLP semantics. Adding an oracle expressible with the existing *UES* vocabulary requires only specifying

its transformation templates and constraints; we discuss the onboarding effort and its limitations in Section 6.

3.1.3 Equivalence of Stochastic Rule Instantiation. To justify that stochastic generation does not compromise equivalence in test oracle representation, we characterize the equivalence condition of *UES* rules at the instantiation level. Given a *UES* rule $r = \langle T_{src}, T_{dst}, C \rangle$ and a database instance $\mathcal{D}$, let $[\![Q]\!]_{\mathcal{D}}$ denote the evaluation result of query/algebra expression Q under SQL semantics. Our equivalence argument relies on two conditions.

Template-Equivalence Condition. For any symbolic binding β that satisfies the deterministic guards in C, the instantiated templates remain semantically equivalent on every database instance:

$$[\![\beta(T_{src})]\!]_{\mathcal{D}} = [\![\beta(T_{dst})]\!]_{\mathcal{D}}.$$

This property is established at rule-design time: T_{src} and T_{dst} are derived from the same relational intent, and only equivalence-preserving rewrites are allowed in the template pair. Therefore, before any stochastic choice is made, each rule already defines an equivalence-preserving rewrite skeleton.

Valid-Instantiation Condition. An instantiation θ is valid if all clauses in C are satisfied. For a stochastic clause $e :=$ `Find<cat>`(R), the generator samples an AST fragment according to:

$$e \sim \mathcal{G}_{cat}(\theta(R), \Sigma, \mathcal{D}),$$

The generated fragment e must have the requested AST category, reference only columns visible from $\theta(R)$, and satisfy the corresponding type and scope requirements. Moreover, the extended binding must satisfy every applicable clause $c \in C$, namely $c(\theta[e]) = \text{true}$. These clauses may constrain attribute containment or equality (`AttrsSub`, `AttrsEq`), relationships between expressions or predicates (`ExpressionEq`, `PredicateEq`), and relation or data properties (`TableEq`, `Unique`, `NotNull`). Volatile or nondeterministic constructs are excluded when prohibited by C.

3.2 Lineage-Aware Query Rewriting

Ensuring the semantic correctness and equivalence is a core requirement in query rewriting. Existing test oracle methods, such as TLP and PQS, typically employ single-step transformations with shallow AST depth and predictable structure, where string substitution or local hard-coded rules are often sufficient to preserve semantic correctness and equivalence. However, when multiple test oracles are chained, the rewriting process involves deeper structural modifications that not only introduce column reference ambiguities and scope violations, but also break the consistency of data lineage across intermediate query representations. Consequently, it becomes difficult to preserve semantic equivalence throughout the transformation chain.

To address these issues, we propose a lineage-aware query rewriting method that ensures reference correctness and semantic equivalence in chained rewriting by precisely tracking both table and column lineage. For both relation sources and their exposed columns, OraclePlate maintains lineage information across query scopes and synchronizes it according to scope dependencies during rewriting. For example, in `SELECT q.c1, q.total FROM (SELECT t.c0 AS c1, SUM(t.c2) AS total FROM t GROUP BY t.c0) AS q`, `q.c1` is a projection alias propagated from the base column `t.c0`, whereas `q.total` is a computed aggregate. When the subquery structure is rewritten, the former may need to be rebound to its base-column definition, while the latter must remain bound to the aggregate computation and cannot be replaced by an input column. Thus, column references are more fragile than relation references because their names, scopes, and semantic definitions can evolve independently. OraclePlate therefore uses persistent lineage identifiers to preserve their bindings across rewrites.

3.2.1 Column Lineage Tracking. In deeply nested structures resulting from chained rewriting, column names are inherently fragile as reference bases and are prone to failure due to alias overriding or scope changes. Compared with table references, column references are more fragile in chained rewriting, because they may correspond not only to base columns, but also to aliased projections, composite expressions, aggregate results, window function results, and values propagated through nested subqueries. These cases are more easily affected by alias overriding, scope changes, and structural rewrites. To address this issue, we propose a column lineage tracking method based on a persistent Lineage Identifier (LID), which establishes a stable reference mechanism independent of the SQL syntactic representation. For base-column references, LIDs are assigned according to physical column identity rather than individual AST occurrences. For example, references to the same physical column in different CTEs share the same LID, while scope information distinguishes their respective query contexts.

Definition of LID. During the parsing phase, OraclePlate assigns a globally unique persistent identifier to each base column in the original query AST. The LID serves as the semantic identity credential of the column, remaining immutable throughout the rewriting process. A derived column inherits its source LID if and only if its definition directly passes through an existing column, whereas a column whose definition produces a new semantic value is assigned a fresh LID. This means that regardless of how the column name changes at the SQL text level, OraclePlate maintains its connection to the original data source through the LID.

For example, for the query `SELECT t1.c1 FROM (SELECT c0 AS c1 FROM t0) AS t1`, although the column name undergoes renaming from `c0` to `c1` and a switch in table alias, `t0.c0` and `t1.c1` consistently share the same LID. This design decouples the semantic identity of the column from its syntactic representation, enabling OraclePlate to accurately locate the source of the column across multi-layer nested structures and alias mappings, thereby avoiding reference failures caused by alias substitution or scope changes.

Lineage Tracking Algorithm. OraclePlate implements lineage tracking by maintaining a complete query scope that stores all available table aliases and their corresponding column information within the current query level. When the source of an expression is required, OraclePlate starts from the current scope and searches layer by layer across scopes until the original definition of the column is found.

In chained rewriting, a column may appear in various forms, such as a base-column reference, an aliased projection, an aggregate result, or a value propagated through nested queries. Although these forms differ syntactically, their correctness ultimately depends on the correct resolution of the underlying column references. To accurately identify the origin of data associated with a column-level structure, OraclePlate tries to trace the source of each column reference layer by layer with Algorithm 1. Specifically, OraclePlate initializes an empty lineage set and iterates over all column references in the input structure (Lines 1-2). For each column reference, OraclePlate sets the starting point to the current query scope (Lines 3-4) and begins an upward search across the scope hierarchy (Line 5). Within each scope layer, OraclePlate attempts to find the actual definition V of the current target column (Line 6). When a definition is found, OraclePlate checks whether the column has undergone any transformation by comparing its LID with that of the current column (Lines 7-12). If V shares the same LID as the current column, the value is propagated without semantic transformation, so OraclePlate continues tracing from its source column (Lines 8-9). Conversely, if the LIDs differ, indicating that the definition produces a new semantic value under the LID assignment strategy, OraclePlate marks this definition as the lineage endpoint, adds it to the result set, and halts the upward tracing process for the current column (Lines 10-12). If the definition is not resolved in the current scope, the search proceeds to its immediate parent scope

Algorithm 1: Column Lineage Tracking

Input: Column-Level Structure E, Initial Query Scope S
Output: Lineage Column Set L

```
1  L ← ∅;
2  foreach column reference C in E do
3      current_col ← C;
4      scope ← S;
5      while scope ≠ null do
6          V ← FindDefinition(current_col, scope);
7          if V ≠ null then
8              if V.LID = current_col.LID then
9                  current_col ← V.source;
10             else
11                 L ← L ∪ {V};
12                 break;
13         scope ← scope.parent;
14 return Sort(L);
```

(Line 13). After all references are processed, OraclePlate sorts and returns the resulting lineage column set (Line 14).

3.2.2 AST Node Synchronization Based on Topological Sorting. In SQL rewriting, the validity of column references depends not only on local transformations of individual AST nodes but also on the dependency relationships across query scopes. In nested queries, an outer query often relies on the output columns exposed by an inner query. Therefore, it is necessary to first determine the processing order among scopes before synchronizing node attributes. To this end, OraclePlate constructs a scope graph based on the dependency relationships between parent queries and subqueries and applies topological ordering to determine the synchronization order of scopes. Within each scope, AST attributes are propagated bottom-up to ensure valid SQL generation.

Topological Ordering of Query Scopes. We take Figure 6 as an example to illustrate the process. In the original query, the outer query references column `t2.c1` from subquery alias `t2`. Therefore, the outer scope cannot be synchronized until the output of the inner scope has been resolved. Accordingly, OraclePlate topologically orders the scope dependency graph so that inner scopes are processed before outer scopes, allowing column provenance to be resolved before dependent outer references are rewritten.

Attribute Synchronization and Scope Maintenance. After the scope order is determined, OraclePlate performs bottom-up attribute synchronization over AST nodes within each scope. For example, in Figure 6, OraclePlate first processes node 2, the subquery `SELECT c0, c1 FROM t1 WHERE c2 = 'value'`, and resolves its output columns and provenance, determining that `t2.c1` refers to `t1.c1`. It then processes node 1 in the outer query and updates all references that depend on the subquery output. During this process, nodes such as `Filter`, `Limit`, and `Sort` directly inherit the output attributes of their child nodes, whereas nodes such as `Projection`, `Join`, `Set Operation`, and subquery rewriting nodes require OraclePlate to recompute output columns and their provenance.

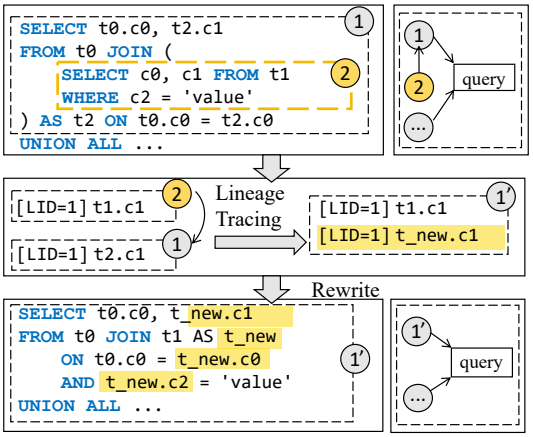


Fig. 6. Bottom-up synchronization of logical attributes using topological sorting.

In this example, the subquery is rewritten into a direct join with base table `t1`, renamed as `t_new`. Accordingly, the outer reference `t2.c1` is updated to `t_new.c1`, the related join predicates are rewritten consistently, and the filter predicate inside the subquery is merged into the outer join condition. To keep all rewritten references resolvable, OraclePlate maintains a scope chain during synchronization. Each query block is associated with an independent scope, and each child scope is linked to its parent scope. When processing a subquery, OraclePlate creates a new scope and registers its output into the parent scope once the subquery has been processed. In this way, renamed aliases and rewritten column references can always be resolved within the correct scope, providing reliable column information for subsequent SQL generation.

3.3 Path-Guided Cascaded Oracle Composition

While isolated test oracles have proven effective at finding specific types of bugs, many deep logic bugs in modern DBMSs emerge only under highly nested and structurally complex queries. OraclePlate overcomes this limitation by composing multiple test oracles into a cascaded rewriting process. Although the process substantially enlarges the testing space, it inevitably introduces the challenge of *path explosion*. Unrestricted random chaining can lead to redundant transformations, such as repeatedly rewriting similar subqueries or revisiting previously explored branches, thereby wasting computational resources and reducing testing efficiency. To address the challenge, OraclePlate applies a rule path-guided strategy to continuously explore new and diverse rewriting paths for each query.

Cascaded Rewriting. OraclePlate is designed to sequentially apply multiple *UES* rules to a single original query. Each application of a *UES* rule produces a rewritten query whose AST remains well formed, allowing the result of one step to directly serve as the input to the next step. Specifically, given an initial query Q_0, a chained transformation of length k produces a sequence of intermediate queries $[Q_1, Q_2, \ldots, Q_k]$. At each step i, OraclePlate selects an applicable rule r_i and applies it to a specific node within the AST of Q_{i-1} to generate Q_i. As illustrated in Figure 7(a), a single query may admit multiple valid match positions for different rules, spanning both outer and nested query blocks. The lineage-aware rewriting in Section 3.2 ensures that each intermediate query remains valid and equivalent and can serve as the input to the next rewriting step.

Unique Rule Paths. To systematically explore the transformation space and mitigate path explosion, OraclePlate employs a path-guided scheduling strategy based on position-aware rule paths. Directly recording the sequence of applied rules (e.g., TLP → PQS) is insufficient to capture

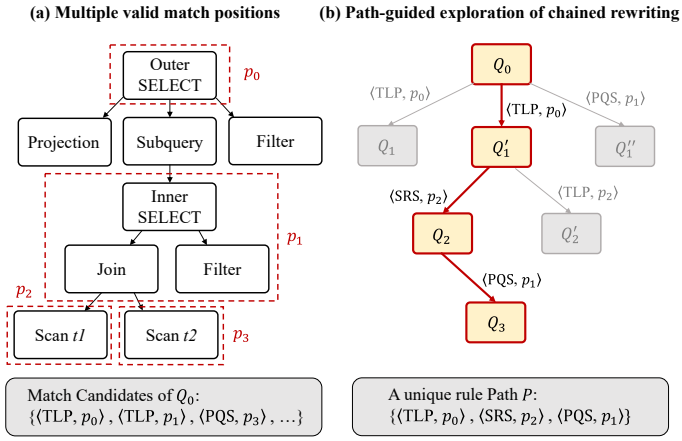


Fig. 7. The process of multi-position matching and cascaded rewriting.

the semantic diversity of the rewritten queries, as applying the same rule at different AST positions can lead to fundamentally different query structures and execution behaviors. For instance, applying a transformation rule to the outermost SELECT clause yields a fundamentally different execution plan than applying the exact same rule to a deeply nested subquery. Therefore, we incorporate the *position* of the rewritten AST node into the path definition.

Formally, we define a transformation step as a tuple $s = \langle r, p \rangle$, where r is the applied *UES* rule and p is the relative AST position of the modified node. A unique rule path P of length k is defined as the ordered sequence of these steps:

$$P = [s_1, s_2, \ldots, s_k] = [\langle r_1, p_1 \rangle, \langle r_2, p_2 \rangle, \ldots, \langle r_k, p_k \rangle] \tag{1}$$

Figure 7(b) shows an example in which different choices of $\langle r, p \rangle$ induce different rewriting branches, while the highlighted branch corresponds to one unique rule path.

Path-Guided Exploration. During the continuous fuzzing process, OraclePlate records the unique rule paths explored for each original query. When determining the next transformation step for a query Q_i, OraclePlate generates a set of candidate steps $S_{candidates} = \{\langle r, p \rangle\}$, representing all applicable rules at all valid AST positions within Q_i.

Instead of selecting a candidate uniformly at random, OraclePlate calculates the resulting path $P_{next} = P_{current} \oplus \langle r, p \rangle$ for each candidate. The scheduler then prioritizes the candidates that form an *unvisited* unique rule path. If all immediate next paths have been visited, the scheduler favors the path with the lowest historical selection frequency. By using the position-aware unique rule path as a feedback mechanism, OraclePlate reduces redundant rewriting loops and encourages exploration of less frequently visited branches. This path-guided strategy improves coverage of the combinatorial rewriting space and increases the likelihood of generating the complex query structures needed to expose deep DBMS logic bugs within a limited testing budget.

4 Implementation

We implemented OraclePlate in Python, consisting of approximately 29 KLoC. The following presents the implementation details.

UES. To make test oracles declarative and composable, OraclePlate applies ANTLR4 [3] to parse *UES* rules and compiles them into an internal representation containing source templates, destination templates, and constraints. Table 3 shows the rewriting rules currently implemented in *UES*, which are drawn from two complementary sources. First, it incorporates metamorphic

Table 3. Rewriting rules currently implemented in *UES*

Test Oracle	Rule Count	Relational Algebra	Description
TLP	8	$\sigma_p(R) \cup_{\text{ALL}} \sigma_{\neg p}(R) \cup_{\text{ALL}} \sigma_{\text{IsNull}(p)}(R) \equiv R$	Query-partitioning rewrites that split a query into three-valued branches while preserving equivalence.
PQS*	1	$R' = \sigma_{\phi_{\text{pivot}}}(R),\ t_{\text{pivot}} \in R'$	Pivot-preservation rewrites that constrain the transformed query to retain the designated pivot row.
EET	2	$Q[e] \Rightarrow Q[e'],\ \forall t:\ e(t) = e'(t)$	Expression-level equivalence rewriting with redundant expression branches.
SRS	8	$Q_1(R \bowtie_\theta T) \sim Q_2(R \bowtie_{\theta'} T)$	Join-synthesis rewrites that transform join forms to construct equivalent or comparable result sets.
EDC*	1	$\langle D, Q\rangle \equiv \langle D', Q'\rangle$	Equivalent data construction with consistency checks between original and transformed query-data pairs.
NoREC*	1	$\text{count}(\sigma_e(R)) \equiv \text{sum}(e \wedge \text{IsNotNull}(e), R)$	Count-preserving rewrites that validate equivalence by comparing row counts.
Argus	2	$[[\text{CAQ}_s]] = [[\text{CAQ}_t]]$	Constraint-aware abstract query pair instantiation for logic bug detection.
Rule1	1	$\Pi_{a_1}(\sigma_e(R)) \equiv \sigma_e(\Pi_{a_1}(R))$	Operator-structure reordering that swaps or repositions projection and filter.
Rule2	3	$Q[e] \Rightarrow Q[e'],\ \forall t:\ e(t) = e'(t)$	Transform boolean or arithmetic predicate forms while preserving filtering semantics.
Rule3	3	$\sigma_e(A \odot B) \equiv \sigma_e(A) \odot \sigma_e(B)$	Set-operator distribution rewrites that push filters into both branches of set operations.
Rule4	1	$R \equiv R \cup_{\text{ALL}} \sigma_{\text{false}}(R)$	Dead-branch injection rewrites that add provably empty branches while preserving result equivalence.
Total	31	—	Core oracles plus custom composed rules used by OraclePlate.

Rule1, Rule2, Rule3, and Rule4 are custom rules introduced by OraclePlate, capturing basic equivalence-preserving transformations such as predicate pushdown.
Test oracles marked with * are supported by instantiating their underlying equivalence relations through adapted concrete transformations.

relations developed by existing DBMS test oracles, such as TLP and EET, which capture different bug-revealing patterns. Second, the custom rules (i.e., Rule1–Rule4) are derived from relational algebra equivalences widely used in query optimization, such as predicate pushdown, to provide additional query rewrite structures. Besides equivalent metamorphic relations, OraclePlate is also implemented to support non-equivalent relations like NoREC [27] by identifying their underlying equivalence relations. For NoREC, this relation is captured by row-count preservation across the original and transformed queries, which allows it to be encoded as an equivalence-based rule. Bugs related to transaction isolation, configuration-dependent behavior, or nondeterministic execution are outside our scope.

Cascaded Rewriter. OraclePlate builds on SQLglot [35] to parse input queries into dialect-aware ASTs, normalize query structure, and regenerate executable SQL after rewriting. Based on it, we further implement customized passes for rule matching, query-scope construction, LID propagation, lineage resolution, and topological synchronization. At runtime, OraclePlate enumerates all applicable *UES* rules over valid AST positions, instantiates `Find` operators using schema-aware and type-aware generators, and generates semantically equivalent query variants. The execution manager records the unique rule path $\langle r, p\rangle$ sequence for each generated query to prioritize unvisited rewriting paths.

SQL Generator. OraclePlate is designed as a general framework that can be integrated with different SQL generators. For supporting compositions of various oracles, our implementation provides a general-purpose SQL generator based on SQLsmith [36]. The generator produces structurally diverse seed queries with sufficient syntactic and semantic variability, while remaining independent of any particular oracle logic. The generator produces queries with nesting depths of 2–4 to facilitate subsequent rule composition. This range provides sufficient structures and candidate sites for rewriting while keeping rule matching and query execution manageable. Overly simple queries offer limited rewrite opportunities, whereas overly complex queries may restrict applicable rewrite sites and introduce unnecessary testing overhead.

Case Reducer. Since OraclePlate detects logic bugs through mismatches between queries that satisfy a metamorphic relation, independently reducing the source and target queries may break this relation. Therefore, OraclePlate implements an AST-based reducer that structurally simplifies the source query while preserving the dependencies required by the original *UES* rewrite sequence, and regenerates the target query by replaying the retained rewrite sequence on the reduced source query. A candidate reduction is accepted only when the original rewrite sequence can still be applied to produce valid queries and the resulting query pair continues to exhibit the mismatch. Otherwise, the candidate is discarded.

5 Evaluation

In this section, we evaluate OraclePlate to answer the following research questions:

- **RQ1:** Can OraclePlate detect previously unknown bugs in real-world DBMSs?
- **RQ2:** Can OraclePlate outperform state-of-the-art DBMS logic bug testing approaches?
- **RQ3:** How effective are chained rewriting and rule-guided exploration strategies?

5.1 Evaluation Setup

Tested DBMSs and Compared Approaches. To evaluate the effectiveness and generality of OraclePlate, we conducted experiments on six widely used real-world DBMSs, including PostgreSQL [25] 18.4, DuckDB [1] 1.5.4, MySQL [23] 9.6.0, MariaDB [22] 12.2.2, TiDB [24] 8.5.5 and Percona [7] 8.0.45. We selected these DBMSs because they are mature, production-grade DBMSs that have been extensively used and tested in both practice and prior research. Moreover, they cover multiple representative implementation styles. MySQL and MariaDB are traditional relational client-server DBMSs, while Percona and TiDB represent distributed DBMSs designed for scalable deployments. We compared OraclePlate with PQS, TLP, EET, and SRS, which are four representative state-of-the-art approaches for DBMS logic bug detection.

Basic Setup. All experiments were conducted on a 64-bit Ubuntu 20.04 server equipped with 128 CPU cores (AMD EPYC 7742 @ 2.25 GHz) and 488 GiB of main memory, with all evaluated DBMSs deployed and tested inside Docker containers. For each target DBMS, every fuzzing campaign was executed for 24 hours, and each Docker container was configured with 8 CPU cores and 50 GiB of RAM for testing.

5.2 DBMS Logic Bug Detection

Bug Statistics. The six tested DBMSs are widely used and have been extensively tested, making previously unknown bugs increasingly difficult to discover. Nevertheless, OraclePlate still detected 39 previously unknown bugs during continuous testing. As shown in Table 4, OraclePlate found 2, 3, 9, 13, 7, and 5 bugs in PostgreSQL, DuckDB, MySQL, MariaDB, TiDB, and Percona, respectively. All 39 bugs have been confirmed by developers, and 9 have already been fixed. The detected issues include 35 logic bugs and 4 crashes.

Table 4. List of previously unknown bugs detected by DBMS testing with OraclePlate employed

#	DBMS	ID	Status	Type	Severity	Rules
1	PostgreSQL	19588	Confirmed	Logic Bug	-	*EET* → Rule3 → *EET*
2	PostgreSQL	19589	Confirmed	Logic Bug	-	Rule2 → Rule3 → *SRS* → *TLP*
3	DuckDB	#24328	Fixed	Logic Bug	-	*SRS* → *EET* → *Argus* → Rule1
4	DuckDB	#27144	Fixed	Logic Bug	-	PQS^* → $NoREC^*$ → Rule3
5	DuckDB	#24329	Fixed	Crash	-	*TLP* → *TLP* → *SRS* → Rule1
6	MySQL	120094	Confirmed	Logic Bug	Critical	*EET* → Rule1 → *TLP* → *TLP*
7	MySQL	120097	Confirmed	Logic Bug	Critical	*SRS*
8	MySQL	120098	Confirmed	Logic Bug	Critical	*EET* → *TLP* → *SRS* → *SRS*
9	MySQL	120099	Confirmed	Logic Bug	Critical	*EET* → Rule1 → Rule1 → *EET*
10	MySQL	120106	Confirmed	Logic Bug	Critical	*Argus* → Rule1
11	MySQL	120265	Confirmed	Logic Bug	Critical	PQS^* → Rule3
12	MySQL	120263	Confirmed	Logic Bug	Serious	*EET* → PQS^*
13	MySQL	120269	Confirmed	Logic Bug	Critical	PQS^* → PQS^* → Rule3
14	MySQL	120270	Confirmed	Logic Bug	Critical	PQS^* → PQS^* → Rule3
15	MariaDB	MDEV-39103	Confirmed	Logic Bug	Minor	Rule1 → *TLP*
16	MariaDB	MDEV-39111	Fixed	Logic Bug	Major	*Argus* → *EET*
17	MariaDB	MDEV-39112	Fixed	Logic Bug	Major	*EET* → *EET*
18	MariaDB	MDEV-39114	Confirmed	Logic Bug	Major	*Argus* → Rule1
19	MariaDB	MDEV-39117	Confirmed	Logic Bug	Minor	*Argus* → *TLP*
20	MariaDB	MDEV-39313	Confirmed	Logic Bug	Major	Rule2 → *SRS* → *TLP*
21	MariaDB	MDEV-39314	Confirmed	Logic Bug	Major	PQS^* → *SRS* → Rule1 → *EET*
22	MariaDB	MDEV-39323	Confirmed	Logic Bug	Critical	*SRS* → Rule2 → Rule4 → *TLP*
23	MariaDB	MDEV-39363	Fixed	Logic Bug	Major	*TLP*
24	MariaDB	MDEV-41517	Confirmed	Logic Bug	Critical	PQS^* → PQS^* → Rule3
25	MariaDB	MDEV-38959	Confirmed	Crash	Major	Rule4 → EDC^*
26	MariaDB	MDEV-38960	Confirmed	Crash	Critical	*EET* → $NoREC^*$
27	MariaDB	MDEV-39180	Confirmed	Crash	Critical	*SRS* → *SRS* → *EET*
28	TiDB	#67330	Fixed	Logic Bug	Critical	*SRS* → *EET* → *TLP* → *TLP*
29	TiDB	#67348	Fixed	Logic Bug	Critical	Rule2 → *EET* → Rule1 → *TLP*
30	TiDB	#67373	Fixed	Logic Bug	Major	*SRS* → Rule4 → *TLP*
31	TiDB	#67735	Confirmed	Logic Bug	Major	*SRS* → Rule1 → *TLP* → Rule2
32	TiDB	#67737	Confirmed	Logic Bug	Major	*SRS*
33	TiDB	#71297	Confirmed	Logic Bug	Critical	*TLP* → *EET*
34	TiDB	#71298	Confirmed	Logic Bug	Critical	*TLP*
35	Percona	PS-11030	Confirmed	Logic Bug	Critical	PQS^* → PQS^* → Rule3
36	Percona	PS-11034	Confirmed	Logic Bug	Critical	PQS^* → Rule3
37	Percona	PS-11035	Confirmed	Logic Bug	Critical	*EET* → Rule1 → Rule1 → *EET*
38	Percona	PS-11040	Confirmed	Logic Bug	Critical	*EET* → PQS^*
39	Percona	PS-11041	Confirmed	Logic Bug	Critical	PQS^* → Rule3

We analyzed the rule chains associated with the detected bugs. Among the 39 confirmed bugs, 35 are exposed only through composition, with their triggering conditions involving cross-semantic interactions, such as those between predicate evaluation and relational transformations. Their rewriting chains range in depth from 2 to 4, suggesting that many DBMS logic bugs can be reached only through deep rewriting paths that are inaccessible to individual test oracles. To make these bug-triggering cases easier to analyze and report, we applied the case reducer to the corresponding queries. It reduced the number of AST nodes by 78.9% on average, while preserving the original rule chains and result mismatches.

Bug Severity. We further examine the severity of the reported logic bugs across different DBMSs. These bugs can have broad consequences because errors in fundamental query processing may propagate to upper-layer queries and applications. Among the 39 bugs confirmed by developers, 32

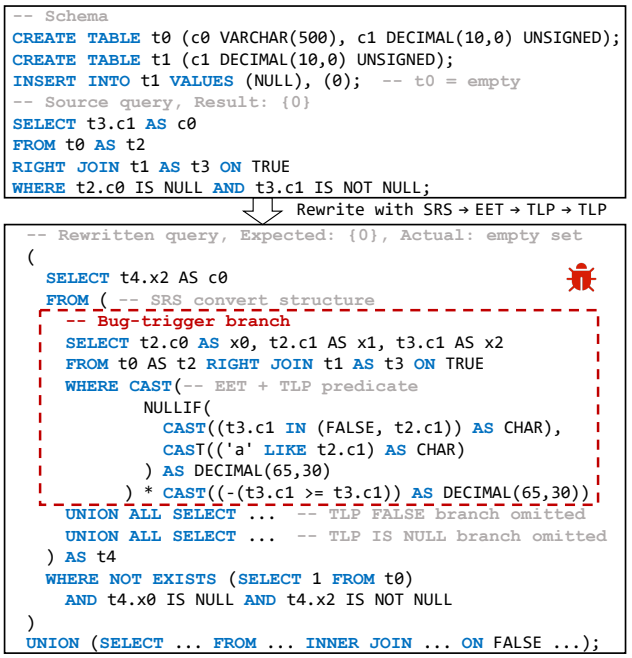


Fig. 8. A case to trigger bug TiDB #67330. It is found via the rule sequence SRS → EET → TLP → TLP.

were classified as critical, serious or major because they affect query correctness or system stability. In particular, several bugs exposed by OraclePlate violate core SQL semantics under chained rewriting, causing valid queries to silently return incorrect results. Such issues are particularly severe because they are difficult to detect in routine use and undermine the trustworthiness of query results.

Case Study 1. Figure 8 presents a representative TiDB bug exposed only by a chained rewriting sequence. The source query returns {0}, but the query obtained through SRS → EET → TLP → TLP incorrectly returns an empty set. The bug appears only after multiple semantics-preserving rewrites are composed. After reduction, we observed that the chain gradually reshapes the query into a rare but valid form. In particular, the final rewriting introduces a compound predicate involving `NULLIF`, `CAST`, and arithmetic operations. TiDB fails to preserve SQL three-valued logic in the presence of NULL values under this form, causing a surviving row to be filtered out incorrectly. This case demonstrates that some logic bugs are inherently interaction-driven and cannot be exposed by isolated oracle applications.

Case Study 2. Figure 9 presents a representative MySQL bug exposed by the rewriting sequence EET → Rule1 → Rule1 → EET. The source query returns the empty set, but the rewritten query incorrectly returns {(0,0)}. No inconsistency is revealed by the original query or by any isolated rewrite. The failure appears only after the query is transformed into a nested form. After reduction, we found that the two consecutive applications of Rule1, which push or pull a `Filter` across a `Projection`, relocate the original emptiness constraint into deeper query blocks and eventually rewrite it into a contradictory predicate. Under this form, MySQL fails to preserve the original branch's emptiness, allowing a row that should be filtered out to survive. This bug stems from incorrect preservation of predicate equivalence across nested query boundaries.

```
-- Schema
CREATE TABLE t0 (c0 INT UNSIGNED); -- empty branch
CREATE TABLE t1 (c0 INT(180) UNSIGNED);
INSERT INTO t1 VALUES (0);
-- Source query, Result: empty set
SELECT ... FROM (
  SELECT c0 AS c0, c0 AS c1 FROM t1 WHERE FALSE
) AS s0 WHERE s0.c0 IS NOT NULL;
```

Rewrite with EET → Rule1 → Rule1 → EET

```
-- Rewritten query, Expected: empty set, Actual: {(0,0)}
SELECT ... FROM (
  -- Rule1 #1
  SELECT ... FROM (
    -- Rule1 #2
    SELECT ... FROM t1
    WHERE -- EET #2
      c0 <> CAST(1 / (-87.45) AS CHAR)
      AND NOT (c0 <> CAST(1 / (-87.45) AS CHAR)) AND ...
  ) AS t_inner
  WHERE t_inner.c0_1 IS NOT NULL AND ( ... )  -- EET #1
) AS t_outer;
```

Fig. 9. A case to trigger the bug MySQL #120099. It is found via the rule sequence EET → Rule1 → Rule1 → EET.

Why these bugs are only detected by OraclePlate. Logic bugs whose triggering conditions arise from interactions among multiple semantic operations can be exposed through oracle composition but remain unreachable to isolated oracle applications. The effectiveness of OraclePlate lies in its ability to compose heterogeneous test oracles within a unified *UES* representation, enabling semantics-preserving rewriting chains across oracle boundaries. Such chains can progressively construct query forms whose structural properties emerge only after multiple rewrites, thereby exercising subtle interactions that single-step or isolated transformations are unlikely to reach. Consequently, OraclePlate can expose bugs triggered by the accumulated effects of multiple transformations while preserving correctness throughout the rewriting process.

5.3 Comparison with Other Testing Approaches

We compare OraclePlate against state-of-the-art logic-bug testing approaches, including TLP [28], EET [16], PQS [29], and SRS [17]. We evaluate them on bug detection capability and branch coverage under the same 24-hour testing budget.

Branch Coverage. Table 5 compares branch coverage under the same 24-hour budget. OraclePlate finds a total of 545,288 branches, compared with 518,334, for TLP, 350,201 for EET, 374,704 for PQS, and 406,207 for SRS. The relative improvements are 5.2%, 22.5%, 14.5%, and 5.6%, respectively (excluding unsupported DBMS). At the DBMS level, OraclePlate also achieves the highest coverage on all six systems. These results suggest that cascaded oracle composition expands the space of semantically valid follow-up queries and drives execution into less-tested paths, which helps to find deep logic bugs.

Bug Detection Capability. Table 6 reports the number of bugs detected within 24 hours. OraclePlate finds 31 bugs, including 2, 3, 6, 10, 5, and 5 in PostgreSQL, DuckDB, MySQL, MariaDB, TiDB, and Percona, respectively. In comparison, TLP finds 6 bugs and SRS finds 8, while EET and PQS find none on their supported DBMSs. Overall, OraclePlate detects 25, 28, 28, and 20 more bugs than TLP, EET, PQS, and SRS, respectively. OraclePlate outperforms other testing approaches on all evaluated DBMSs, with the largest margin on MariaDB (+9), indicating that the advantage of oracle composition is consistent across engines.

Table 5. Comparison of Code Coverage by Different Approaches in 24 Hours (excluding unsupported DBMS)

DBMS	OraclePlate	TLP	EET	PQS	SRS
Postgres	54,622	51,608	23,186	44,707	52,679
DuckDB	116,278	115,025	-	-	-
MySQL	118,734	108,433	96,431	108,655	112,428
MariaDB	59,621	54,410	58,486	51,282	54,741
TiDB	83,770	80,644	73,883	62,371	79,403
Percona	112,263	108,214	98,215	107,689	106,956
Total	545,288	518,334	350,201	374,704	406,207
Increment	–	5.2%↑	22.5%↑	14.5%↑	5.6%↑

To isolate the effect of oracle composition from that of seed generation, we also enable each oracle family separately within OraclePlate. These oracle-level variants share the same oracle-independent SQLsmith generator and implementation pipeline; only the enabled rewriting rules differ. Under this controlled setting, OraclePlate detects 31 bugs, compared with 3 for TLP, 0 for EET, 0 for PQS, and 2 for SRS.

PQS and EET detect no bugs in our experimental evaluation. One possible reason is that the target DBMS may already have been tested with these tools. Since both tools are open source and the DBMSs themselves are widely used, they are regularly and thoroughly tested by both the community and the database developers.

Figure 10 presents the Venn diagrams of bugs found by OraclePlate, TLP, and SRS. Figure 10(a) shows that all bugs found by TLP are also detected by OraclePlate, which is expected because OraclePlate incorporates TLP. Beyond these shared bugs, OraclePlate detects 25 additional bugs. Figure 10(b) compares OraclePlate with SRS. OraclePlate detects 23 bugs not found by SRS, whereas SRS detects 3 bugs not found by OraclePlate. This discrepancy mainly arises from limitations of the current general generator and related implementation factors. For example, the current generator does not produce connection operators such as Semi Join and Anti Join, through which SRS is able to trigger bugs.

5.4 Contributions of Key Approaches

We evaluate two approaches in OraclePlate separately: (1) chained rewriting with different depths and (2) path-guided scheduling.

Table 6. Comparison of Bugs Detected by Different Approaches in 24 Hours (excluding unsupported DBMS)

DBMS	OraclePlate	TLP	EET	PQS	SRS
Postgres	2	0	0	0	0
DuckDB	3	0	-	-	-
MySQL	6	2	0	0	2
MariaDB	10	1	0	0	1
TiDB	5	1	0	0	4
Percona	5	2	0	0	1
Total	31	6	0	0	8
Increment	–	+25	+28	+28	+20

Table 7. Comparison of guided and unguided rewriting across different DBMSs in 24 hours

DBMS	Bug Count			Coverage			Unique Rule Paths		
	Guided	Unguided	Inc.	Guided	Unguided	Inc.	Guided	Unguided	Inc.
Postgres	2	0	+2	54,622	46,621	17%↑	4,710,258	2,133,539	121%↑
DuckDB	3	1	+2	116,278	93,759	24%↑	5,210,354	2,067,491	152%↑
MySQL	6	2	+4	118,734	91,821	29%↑	5,574,526	2,567,142	117%↑
MariaDB	10	5	+5	59,621	51,698	15%↑	2,783,912	1,093,722	154%↑
TiDB	5	2	+3	83,770	68,754	21%↑	3,016,473	1,200,948	151%↑
Percona	5	3	+2	112,263	89,958	25%↑	5,067,550	2,177,096	133%↑

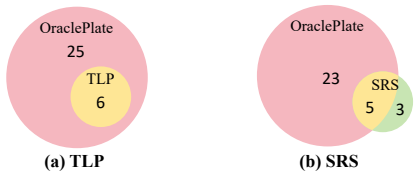


Fig. 10. Venn diagram for bugs found by different tools (excluding unsupported DBMS).

Table 8. Comparison of path-guided OraclePlate (Path) and coverage-guided OraclePlate^g (Cov.) across different DBMSs in 24 hours.

DBMS	Bug Count			Branch Coverage		
	Path	Cov.	Imp.	Path	Cov.	Imp.
PostgreSQL	2	0	+2	54,622	51,489	6%↑
DuckDB	3	3	+0	116,278	103,819	12%↑
MySQL	6	3	+3	118,734	93,491	27%↑
MariaDB	10	7	+3	59,621	54,209	10%↑
TiDB	5	2	+3	83,770	72,235	16%↑
Percona	5	4	+1	112,263	92,424	21%↑
Total	31	19	+12	545,288	467,667	17%↑

Effect of Chained Rewriting. We evaluate the component with different max rewriting depth ranges from 1 to 4. Figure 11 shows that rewriting depth substantially affects bug-finding effectiveness. After 24 hours, depth=4 consistently achieves the highest bug counts across all six DBMSs, with 2, 3, 6, 10, 5 and 5 on PostgreSQL, DuckDB, MySQL, MariaDB, TiDB and Percona. In comparison, depth=1 finds 0, 0, 1, 1, 2, and 0 bugs on these DBMSs, respectively; depth=2 finds 0, 0, 4, 5, 3, and 3; and depth=3 finds 1, 1, 4, 8, 4, and 4. The curves also show that deeper configurations continue discovering bugs in later stages, whereas shallow ones plateau earlier. These results suggest that many logic bugs require multiple semantics-preserving rewrites and that deeper chains can reach query forms inaccessible to single-step or short-chain rewriting.

Effect of Path-Guided Exploration. Table 7 further shows that guided scheduling consistently outperforms unguided exploration across all DBMSs. In bug detection, guided mode improves from 0 to 2 on PostgreSQL, 1 to 3 on DuckDB, 2 to 6 on MySQL, from 5 to 10 on MariaDB, from 2 to 5 on TiDB, and from 3 to 5 on Percona. Overall, guided exploration detects 31 bugs, compared with 13 in the unguided setting (2.38×). The improvement is also reflected in auxiliary metrics: coverage

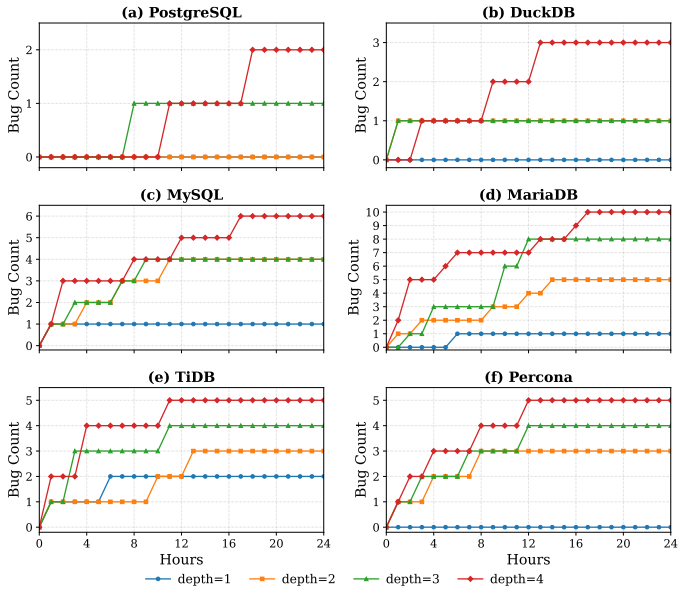


Fig. 11. Comparison of bug detection effectiveness across different rewriting depths in 24 hours.

increases by 15%–29%, and unique rule paths increase by 117%–154%. These results may be due to the path guidance allocating the testing budget across more diverse rewriting branches, allowing it to uncover more bugs.

We implemented OraclePlate^g, which prioritizes rules that trigger new branch coverage, to compare path-guided and coverage-guided exploration. Under the same 24-hour budget across six DBMSs, OraclePlate detected 31 bugs and covered 545,288 branches, compared with 19 bugs and 467,667 branches for OraclePlate^g (Table 8). This gap mainly results from coverage-feedback overhead: instrumentation reduced throughput by 6% on PostgreSQL, 12% on DuckDB, 27% on MySQL, 10% on MariaDB, 16% on TiDB, and 21% on Percona, limiting exploration of deep rewriting chains. We therefore retain lightweight path guidance, while lower-overhead or sampled coverage collection remains a complementary direction.

These results indicate that the two components are complementary. Chained rewriting enlarges the reachable space of semantically equivalent queries, while path-guided scheduling improves search efficiency by prioritizing less-explored rule-position combinations.

6 Discussion

Rule Extensibility and Cross-DBMS Adaptation Cost. Across the currently implemented oracles, *UES* rules average approximately 10 lines of declarative specification. Onboarding a new oracle typically involves two steps, using TLP as an example: 1) Extracting the oracle's metamorphic relation into source (T_{src}) and target (T_{dst}) structures. For TLP, T_{src} is encoded as a projection over an input relation, $\mathsf{Proj}\langle a_0\rangle(\mathsf{Input}\langle r_0\rangle)$, while T_{dst} rewrites it into a union of three guarded projections with conditions e_0, $\neg e_0$, and $\mathsf{IsNull}(e_0)$. 2) Encoding its applicability conditions and symbol bindings in C. In TLP, C generates e_0 from r_0 via $\mathsf{Find}\langle rand\rangle(r_0)$, while $\mathsf{AttrsSub}(a_0, r_0)$ ensures consistent attribute binding.

Once a new equivalence-based metamorphic rule is encoded in the unified *UES* representation, it no longer requires a handwritten rewriter for each individual DBMS. Cross-DBMS porting

mainly requires adapting the SQL generator, the `Find` component, and dialect-specific SQL processing. Oracles requiring new operators or constraint types additionally require extending the *UES* vocabulary and the corresponding validation or generation support.

Extensibility to non-equivalent metamorphic relations is limited to cases whose decision conditions can be preserved through an equivalence-based reformulation using intermediate representations or explicit consistency conditions. For example, NoREC [27] detects bugs by checking whether different query forms produce consistent row counts. Although it does not compare full results, its rewriting relation provides a verifiable consistency condition that can be incorporated into OraclePlate.

Beyond general-purpose relational DBMSs, *UES* can in principle support specialized DBMS domains by extending its rule vocabulary with domain-specific syntax and constraints. For example, supporting AEIs for spatial DBMSs [14] would require spatial types, functions, and conditions such as SRID consistency.

Overhead of Chained Rewriting. Compared with single-rule testing, OraclePlate incurs additional overhead in each testing round due to multi-rule matching, constraint checking, and generic AST-based rewriting. As the rewriting chain grows longer, the system must maintain more complex column-lineage and scope information to ensure that all intermediate queries remain valid. These steps inevitably increase the computational cost of the framework and the processing overhead for each seed query. Our experiments show that SQLancer executes hundreds to thousands of queries per second. In contrast, at depth 4, OraclePlate processes about 160–300 queries per second across the complete testing pipeline, including both equivalent-query generation and DBMS execution, due to the larger rule space and higher constraint validation cost. In our 24-hour experiments, equivalent-query generation accounts for approximately 87% of the total runtime, while DBMS execution accounts for about 13%.

However, this extra cost is justified by the benefit it brings. A sufficiently powerful SQL generator could directly produce some final query forms reached by a rewriting chain. Yet one-shot generation does not provide an equivalent reference query or systematically explore combinations of oracle-specific features. OraclePlate instead applies path-guided oracle composition to generator-produced seeds and checks equivalent intermediate queries, exposing bugs at states that one-shot generation followed by a single comparison rewrite may miss. The same-generator comparison further shows that these gains primarily arise from oracle composition rather than seed-query complexity.

False Positives. During bug detection with OraclePlate, we observed 1 false positive case. Specifically, it involves a table containing a `BIGINT UNSIGNED ZEROFILL` column, where OraclePlate generates two theoretically equivalent queries but the DBMS returns different results. According to developer feedback, this case arises from DBMS-specific handling of the deprecated `ZEROFILL` feature rather than from an error in our rewriting logic. To mitigate this issue, it is possible to exclude deprecated DBMS-specific features and nondeterministic constructs, such as ordering-sensitive operations and nondeterministic functions.

7 Related Work

DBMSs Logic Bug Detection. Logic bugs in DBMSs are hard to detect because they do not have observable symptoms like crashes. Early works like RAGS [33], DQE [34] and DT2 [12] utilize differential testing to test logic bugs. However, finding a suitable comparable DBMS is difficult. PQS [29] uses a designated pivot row as a reference and checks whether the generated query can still retrieve that row. Recent works utilize metamorphic testing [9, 10, 18, 31, 32] to find logic bugs in DBMSs. For example, TLP [28] partitions a query according to the three-valued semantics of a predicate and verifies the consistency between the original query and the union of the

partitioned ones. NoREC [27] rewrites a predicate into a non-optimizable form and checks whether the cardinality of the rewritten query remains consistent. SRS [17] constructs equivalent queries by synthesizing equivalent set relations over JOIN operations. EET [16] performs equivalence-preserving transformations at the expression level. EDC [13] detects logic bugs by constructing equivalent data sets based on equivalence relations.

OraclePlate also utilizes metamorphic testing to test logic bugs. Existing metamorphic testing approaches are typically isolated, each with its own transformation logic and execution interface, which makes them difficult to compose. In contrast, OraclePlate standardizes heterogeneous test oracles into a unified representation *UES* and enables their composition through chained rewriting, allowing multiple equivalence-preserving transformations to work together to expose deeper logic bugs in DBMSs.

Query Rewriting. DBMS communities have long studied equivalent rewriting rules for query optimization [4, 6, 8, 15, 19, 37]. WeTune [37] and GRewriter [15] model semantics-preserving rewrites through formalizing rewriting rules. QueryBooster [4] formalizes query rewrites in an expressive rule language to support reusable and human-centered SQL rewriting. Tailwind [38] uses ALPs to describe replaceable logical subplans, while Calcite [6] expresses optimizer rewrite rules through programmatic manipulation of relational expressions. However, these approaches are designed for optimization rather than testing. Their rules are intended to apply semantics-preserving transformations, rather than to support bug-oriented rewriting in DBMS testing. Some researchers have studied utilizing query rewriting to test DBMSs. For example, EQR [20] generates equivalent queries for GDBMS testing via an abstract syntax graph, but its graph-structured abstraction is not directly suitable for traditional relational DBMS testing. Argus [21] represents a test oracle as an equivalent constrained abstract query pair with placeholders.

However, current rewriters remain query-pair oriented and centered on placeholders, making them less suitable for representing test oracles as finer-grained rewriting rules and composing these oracles. Moreover, existing rewriters mainly target optimizer-oriented logical rewrites, whereas *UES* is designed for metamorphic test oracles. This allows *UES* to express stochastic transformations while supporting reusable rule specification and chained composition within a unified framework.

8 Conclusion

This paper presents OraclePlate, a DBMS logic bug detection framework with cascaded rewriting-based test oracle composition. OraclePlate unifies heterogeneous test oracles with *UES*, supports lineage-aware query rewriting, and uses path-guided exploration to prioritize effective rewriting combinations. Thus, it enables multiple equivalence-preserving transformations to work together to expose logic bugs that are difficult to detect with isolated test oracles. We implemented OraclePlate and evaluated it on six widely used DBMSs. The results show that OraclePlate detected 39 previously unknown bugs, with all confirmed by developers and 9 fixed. Compared with existing DBMS testing approaches, OraclePlate found 20 to 28 more bugs and improved coverage by 5.2% to 22.5% in 24 hours. These results demonstrate that composing test oracles through a unified framework is effective for DBMS logic bug detection.

Acknowledgments

We sincerely thank the reviewers for their valuable suggestions. This work was supported by the National Natural Science Foundation of China (62302256, 62372022 and U25A20425), the National Science and Technology Major Project of China (2025ZD0123704), the Fundamental Research Funds for the Central Universities (JKF-2026052809520), the Beijing Natural Science Foundation (L251040), and the CCF-NSFOCUS Kun-Peng Scientific Research Fund (2025002).

References

[1] [n. d.]. DuckDB GitHub. https://github.com/duckdb/duckdb. Accessed: October 27, 2026.

[2] 2015. Database. https://en.wikipedia.org/wiki/Database Accessed: October 27, 2026.

[3] ANTLR4 Contributors. 2021. ANTLR4. https://github.com/antlr/antlr4 Accessed: October 27, 2026.

[4] Qiushi Bai, Sadeem Alsudais, and Chen Li. 2023. QueryBooster: Improving SQL Performance Using Middleware Services for Human-Centered Query Rewriting. arXiv:2305.08272 [cs.DB] https://arxiv.org/abs/2305.08272

[5] Luciano Baresi and Michal Young. 2001. Test oracles. (2001).

[6] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 221–230. doi:10.1145/3183713.3190662

[7] Avatar Magnus Blåudd. [n. d.]. Percona Website. https://www.percona.com/. Accessed: October 27, 2026.

[8] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Seattle, Washington, USA) *(PODS '98)*. Association for Computing Machinery, New York, NY, USA, 34–43. doi:10.1145/275487.275492

[9] Tsong Yueh Chen, Shing-Chi Cheung, and Siu-Ming Yiu. 2020. Metamorphic Testing: A New Approach for Generating Next Test Cases. *CoRR* abs/2002.12543 (2020). arXiv:2002.12543 https://arxiv.org/abs/2002.12543

[10] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1, Article 4 (Jan. 2018), 27 pages. doi:10.1145/3143561

[11] Carlos Coronel, Steven Morris, and Peter Rob. 2016. *Database systems: design, implementation, and management*. Cengage learning Boston.

[12] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. 2023. Differentially Testing Database Transactions for Fun and Profit. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 35, 12 pages. doi:10.1145/3551349.3556924

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

[14] Ralf Hartmut Güting. 1994. An introduction to spatial database systems. *the VLDB Journal* 3, 4 (1994), 357–399.

[15] Zhe Jiang, Zhaoguo Wang, Haoning Lan, Chuzhe Tang, Haoran Ding, Lefeng Wang, Songyun Zou, Zhuoran Wei, Yongcun Liu, Xiang Yu, et al. 2025. Grewriter: Practical query rewriting with automatic rule set expansion in gaussdb. *Proceedings of the VLDB Endowment* 18, 12 (2025), 4991–5003.

[16] Zu-Ming Jiang and Zhendong Su. 2024. Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 821–835. https://www.usenix.org/conference/osdi24/presentation/jiang

[17] Jinhui Lai, Chi Zhang, Bingyan Li, Chenglin 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. 3, 6, Article 363 (Dec. 2025), 24 pages. doi:10.1145/3769828

[18] Rui Li, Huai Liu, Pak-Lok Poon, Dave Towey, Chang-Ai Sun, Zheng Zheng, Zhi Quan Zhou, and Tsong Yueh Chen. 2024. Metamorphic Relation Generation: State of the Art and Visions for Future Research. arXiv:2406.05397 [cs.SE] https://arxiv.org/abs/2406.05397

[19] Guy M. Lohman. 1988. Grammar-like functional rules for representing query optimization alternatives. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '88)*. Association for Computing Machinery, New York, NY, USA, 18–27. doi:10.1145/50202.50204

[20] Qiuyang Mang, Aoyang Fang, Boxi Yu, Hanfei Chen, and Pinjia He. 2024. Testing Graph Database Systems via Equivalent Query Rewriting. 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 143, 12 pages. doi:10.1145/3597503.3639200

[21] Qiuyang Mang, Runyuan He, Suyang Zhong, Xiaoxuan Liu, Huanchen Zhang, and Alvin Cheung. 2026. Automated Discovery of Test Oracles for Database Management Systems Using LLMs. arXiv:2510.06663 [cs.DB] https://arxiv.org/abs/2510.06663

[22] MariaDB 2026. MariaDB. https://mariadb.org/. Accessed: October 27, 2026.

[23] MySQL 2026. MySQL. https://www.mysql.com/. Accessed: October 27, 2026.

[24] PingCAP. 2026. TiDB. https://github.com/pingcap/tidb/. Accessed: October 27, 2026.

[25] PostgreSQL 2024. PostgreSQL. https://www.postgresql.org/. Accessed: October 27, 2026.

[26] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. 2003. *Database management systems*. Vol. 3. McGraw-Hill New York.

[27] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1140–1152. doi:10.1145/3368089.3409710

[28] 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

[29] 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

[30] Manuel Rigger and Zhendong Su. 2023. SQLancer. https://github.com/sqlancer/sqlancer Accessed: October 27, 2026.

[31] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (2016), 805–824. doi:10.1109/TSE.2016.2532875

[32] Sergio Segura, Dave Towey, Zhi Quan Zhou, and Tsong Yueh Chen. 2020. Metamorphic Testing: Testing the Untestable. *IEEE Software* 37, 3 (2020), 46–53. doi:10.1109/MS.2018.2875968

[33] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. 618–622.

[34] 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

[35] SQLglot Contributors. 2021. SQLglot. https://github.com/tobymao/sqlglot Accessed: October 27, 2026.

[36] SQLsmith Contributors. 2015. SQLsmith. https://github.com/anse1/sqlsmith Accessed: October 27, 2026.

[37] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 94–107. doi:10.1145/3514221.3526125

[38] Geoffrey X. Yu, Ryan Marcus, and Tim Kraska. 2026. Tailwind: A Practical Framework for Query Accelerators. arXiv:2604.28079 [cs.DB] https://arxiv.org/abs/2604.28079

Received 17 April 2026; revised 19 August 2026; accepted 12 September 2026