ValScope: Value-Semantics-Aware Metamorphic Testing for Detecting Logical Bugs in DBMSs

Abstract

Pending...

1 Introduction

2 Background

2.1 Database Management Systems and SQL

Database Management Systems. Database Management Systems (DBMSs) are fundamental to modern software ecosystems, offering systematic mechanisms for storing, organizing, and accessing large volumes of structured data. This work focuses on *relational DBMSs*, which manage data according to the relational model [5]. In such systems, data is organized into tables, each containing tuples (records) that represent real-world entities. Mathematically, each table corresponds to a relation defined over a finite set of attributes, and the database comprises a collection of such relations. In practice, DBMSs allow developers to efficiently insert, update, delete, and query data through structured interfaces, ensuring both data consistency and accessibility.

SQL. Structured Query Language (SQL) [4] is the standard interface for interacting with relational DBMSs. It provides a unified syntax for defining, manipulating, and querying data, and can be broadly classified into four categories: Data Definition Language (DDL), Data Manipulation Language (DML), Data Query Language (DQL), and Data Control Language (DCL). Among them, DQL, represented primarily by the `SELECT` statement, forms the core of data retrieval in relational systems. DQL is used to specify the desired information without explicitly describing how it should be obtained, reflecting SQL's declarative nature. The `SELECT` statement supports rich semantics including filtering, grouping, aggregation, and joining across multiple relations, making it the most fundamental yet semantically complex component of SQL. In this work, we focus on detecting logical bugs in DQL, as they constitute the majority of real-world query workloads and are crucial to ensuring the correctness and reliability of DBMS query processing.

2.2 Logical Bugs in DBMSs

Logical Bugs. Logical bugs are one of the most critical types of bugs in DBMSs, silently causing incorrect query results without triggering system crashes [8, 13]. Unlike crash bugs that exhibit obvious failures, logical bugs corrupt query outputs in subtle ways, posing serious risks to data integrity and application reliability.

Existing Detection Approaches. Existing research on detecting logical bugs mainly relies on automated testing techniques. However, designing an effective testing framework is challenging due to the **test oracle problem**—it is difficult to determine the correct result of a complex SQL query for comparison. To address this issue, researchers have explored three main categories of approaches [8]. The first is differential testing [3, 19], which executes the same query across multiple DBMSs and compares their outputs; inconsistencies reveal potential logic flaws, but dialect differences and heterogeneous semantics often limit its applicability. The second, oracle-guided synthesis [18], generates queries expected to return a specific pivot row and reports an error when the row is missing, but it only captures localized issues and fails to expose deeper semantic inconsistencies. The third and most prominent category is metamorphic testing, which establishes expected semantic relations between pairs of systematically transformed queries. This approach enables *oracle-free* verification, offering better scalability and generality. We will provide an in-depth analysis in the following section.

2.3 Metamorphic Testing

In recent years, metamorphic testing (MT) has become the most effective and widely adopted approach for detecting logical bugs in DBMSs [8, 13]. The core idea of MT is to construct multiple SQL statements whose results are expected to satisfy a specific relation, known as a metamorphic relation (MR). When the actual query results violate this relation, it indicates that at least one query triggers a logical bug in the tested DBMS.

Equivalent MR. Traditional MT, such as NoREC [16] and TLP [17], define the metamorphic relation as strict equivalence between the outputs of the original and transformed queries. For example, NoREC transforms an optimizable query into a non-optimizable one, while TLP decomposes a predicate into multiple subqueries and merges their results to ensure equality. Although this equivalence-based strategy effectively bypasses the lack of ground truth, it constrains the search space of test cases: the transformations typically preserve all operators, functions, and predicates of the original query, merely changing structural forms. As a result, such approaches are limited to verifying surface-level correctness and often fail to reveal deeper semantic errors—particularly those that do not produce directly inconsistent result sets but still violate relational semantics internally [8].

Approximation MR. To overcome the over-restrictive nature of equivalence-based testing, Pinolo [8] introduces the concept of set-semantic approximation relations, relaxing the equivalence assumption. By expanding or constraining query predicates, it generates over- and under-approximate queries and judges correctness through inclusion or containment relations between result sets. This relaxation enables the detection of a broader spectrum of logical bugs, uncovering deeper semantic inconsistencies that equivalence-based approaches often miss [8].

However, approximation at the set level still overlooks subtle semantic shifts that occur at the value level, such as incorrect aggregations, ordering, or numeric deviations that preserve the same tuple set but alter the meaning of the result. Building on this insight, our work extends MT from set semantics to value semantics. We propose a new class of value-semantic approximation relations that reason about the direction and magnitude of result changes while maintaining structural consistency. By integrating set-semantic approximation with value-semantic approximation, our framework establishes a unified, multi-dimensional criterion for logical bug detection—enabling the discovery of more nuanced and deeply hidden semantic faults that existing MT frameworks cannot capture.

3 SQL Query Approximation

Traditional testing frameworks typically rely on strict equivalence or simple set inclusion checks, which are insufficient to capture subtle semantic deviations introduced by optimizer transformations or operator mutations [8]. To address this limitation, we introduce a more comprehensive notion of SQL Query Approximation, which unifies two complementary perspectives of query behavior: the **set-semantic** and **value-semantic** dimensions. The set-semantic dimension characterizes differences in the returned tuple sets, while the value-semantic dimension captures monotonic variations in the computed or aggregated values over those tuples. Together, these two dimensions form a unified framework for expressing and reasoning about semantic consistency, enabling the construction of more expressive test oracles that can detect both structural and value-level inconsistencies in DBMS behavior.

3.1 Set-Semantic Approximation

In this section, we first formalize the notion of approximation at the set level. This relation captures inclusion or containment among query result sets.

Definition 3.1 (Set-Semantic Approximation Relation)**.** Given a database D, let q_1 and q_2 be two SQL queries whose result sets are $R(q_1, D)$ and $R(q_2, D)$, respectively. We say that q_1 is the *set-level under-approximation* of q_2 over D, denoted by $q_1 \preceq_D^s q_2$, if and only if:

$$R(q_1, D) \subseteq R(q_2, D)$$

Conversely, q_1 is the *set-level over-approximation* of q_2 over D, denoted by $q_1 \succeq_D^s q_2$, if and only if:

$$R(q_1, D) \supseteq R(q_2, D)$$

Here, $R(q, D)$ represents the multi-set returned by evaluating query q on database D, and $\subseteq$ and $\supseteq$ denote inclusion and containment relations between two multi-sets.

Intuitively, the set-semantic approximation forms a partial order over queries: $q_1 \preceq_D^s q_2$ means that q_1 produces a narrower or more restrictive result than q_2, while $q_1 \succeq_D^s q_2$ means that q_1 yields a broader or less restrictive result. These two relations are inverses of each other and together define the lattice of set-level approximations.

Example 3.1. Consider a database $D = \{t_1\}$, where $t_1(c_1) = \{-1, 0, 1\}$. Let the following queries be defined:

$$q_1 : \textbf{SELECT } c_1 \textbf{ FROM } t_1 \textbf{ WHERE } c_1 \le 0$$

$$q_2 : \textbf{SELECT } c_1 \textbf{ FROM } t_1 \textbf{ WHERE TRUE}$$

$$q_3 : \textbf{SELECT } c_1 \textbf{ FROM}t_1 \textbf{ WHERE } c_1 < 0$$

We have:

$$R(q_3, D) = \{-1\}, \quad R(q_1, D) = \{-1, 0\}, \quad R(q_2, D) = \{-1, 0, 1\}$$

Hence,

$$R(q_3, D) \subseteq R(q_1, D) \subseteq R(q_2, D)$$

which gives the approximation chain:

$$q_3 \preceq_D^s q_1 \preceq_D^s q_2$$

Intuitively, q_3 is a stricter version of q_1, and q_1 a stricter version of q_2, each progressively expanding the selection condition and thus broadening the result set.

3.2 Value-Semantic Approximation

To overcome the limitation of purely set-level inclusion, we extend the approximation relation from the result-set level to the value level. Unlike the set-level relation that focuses on tuple inclusion, the value-semantic relation captures the monotonic variation of target columns — the columns whose values are directly affected by functional or aggregation operations. This allows the framework to detect logical bugs where queries return identical tuples but diverge in their value semantics, such as incorrect computations in aggregation or updates.

Definition 3.2 (Value-Semantic Approximation Relation)**.** Given a database D, let q_1 and q_2 be two SQL queries whose result sets are $R(q_1, D)$ and $R(q_2, D)$, respectively. Let $C_t \subseteq \mathrm{Cols}(R(q_1, D)) \cap \mathrm{Cols}(R(q_2, D))$ denote the target columns whose values will be compared. Let G denote the grouping or ordering basis, determined as follows:

If the query contains a `GROUP BY` clause, G corresponds to the group-by keys. Otherwise, G represents a deterministic ordering over non-target columns (e.g., primary key or lexicographic ordering of attributes) to align tuples for comparison.

We say that q_1 is the *value-level over-approximation* of q_2 over D, denoted by $q_1 \succeq_D^v q_2$, if and only if:

$$\forall g \in G^*, \forall c \in C_t,\ V_{q_1}(g, c) \ge V_{q_2}(g, c)$$

where G^* is the set of all comparable tuple groups under G, and $V_q(g,c)$ denotes the value of column c in group g (or tuple position) produced by query q.

Conversely, q_1 is the *value-level under-approximation* of q_2, denoted by $q_1 \preceq_D^v q_2$, if and only if:

$$\forall g \in G^*, \forall c \in C_t,\ V_{q_1}(g,c) \leq V_{q_2}(g,c)$$

This definition unifies two cases: group-wise comparison for aggregation queries, and order-aligned comparison for non-aggregated results.

Intuitively, the *set-semantic approximation* ($\preceq_D^s$) describes inclusion of tuples, while the *value-semantic approximation* ($\preceq_D^v$) reflects monotonicity among the values of corresponding tuples.

Example 3.2. Consider a table $t_1(c_1, c_2)$ as follows:

$t_1 =$

c_2	c_1
A	10
A	20
B	5
B	7

Let the following two queries be defined:

$$q_1 : \textbf{SELECT } c_2,\ \text{MAX}(c_1) \textbf{ FROM } t_1 \textbf{ GROUP BY } c_2$$

$$q_2 : \textbf{SELECT } c_2,\ \text{MIN}(c_1) \textbf{ FROM } t_1 \textbf{ GROUP BY } c_2$$

The results are:

$R(q_1, D) =$

c_2	MAX(c_1)
A	20
B	7

$R(q_2, D) =$

c_2	MIN(c_1)
A	10
B	5

Under the grouping basis $G = \{c_2\}$ and target column $C_t = \{c_1\}$, we have for each $g \in G^* = \{\text{A}, \text{B}\}$:

$$V_{q_1}(g, c_1) \geq V_{q_2}(g, c_1)$$

Hence, $q_1 \succeq_D^v q_2$. Intuitively, both queries return identical group sets (thus $q_1 \equiv_D^s q_2$), but differ monotonically in their value semantics: the aggregated value of q_1 in each group is no smaller than that of q_2.

3.3 Approximation Propagation

The approximation relations introduced in the previous section capture the semantic correspondence between two complete SQL queries by comparing their result sets or value outputs. However, in practical DBMS testing, a mutation usually affects only a local part of the query—for instance, a predicate, an operator, or an aggregation function—rather than the entire query. To understand how such a local change influences the final query result, we extend the discussion from the semantic level of full-query comparison to the structural level of SQL. Specifically, we define the concept of *approximation propagation*, which describes how a local approximation relation established at one node of the query's abstract syntax tree (AST) can be transmitted through its parent operators and clauses, thereby determining how a single mutation impacts the overall approximation behavior of the query.

Definition 3.3 (Approximation Propagation)**.** Let D be a database, and let n_1, n_2 denote two semantically comparable nodes (e.g., subqueries, predicates, or expressions) in the SQL AST. We use the unified notation $n_1 \preceq_D^\alpha n_2$ to represent an *approximation relation* of type $\alpha \in \{s, v\}$, where s and v correspond to the set-semantic and value-semantic levels, respectively. The relations defined in §3.1 and §3.2 describe query-level approximations between complete queries. In contrast, approximation propagation extends these relations to the structural level, capturing how local approximations between AST nodes can influence or induce approximations at higher layers of the query.

Formally, each operator op is characterized by two semantic properties: a *mapping* ($\alpha_{in} \rightarrow \alpha_{out}$), which specifies how the operator transforms between set-level and value-level semantics, and a *direction* $\sigma(op) \in \{+1, -1\}$, where $+1$ indicates that the operator preserves the approximation direction (monotone increasing) and -1 indicates that it reverses the direction (monotone decreasing or negating). Based on these properties, the propagation of $n_1 \preceq_D^\alpha n_2$ can be classified into four canonical forms:

- **(Set → Set)**: If a subquery or predicate $p_1 \preceq_D^s p_2$ is embedded under a higher-level set operator op_s (e.g., `EXISTS`, `NOT EXISTS`, logical `NOT`), then the resulting relation satisfies:
$$R(n_1, D) \preceq_D^{s \cdot \sigma(op_s)} R(n_2, D)$$
where operators such as `EXISTS` are monotone increasing ($\sigma = +1$), while `NOT EXISTS` or `NOT` are monotone decreasing ($\sigma = -1$), reversing inclusion ($\subseteq \leftrightarrow \supseteq$).
- **(Set → Value)**: If an aggregation or mapping function f is applied to two relations that satisfy $R(n_1, D) \preceq_D^s R(n_2, D)$, then the corresponding value-level results satisfy:
$$f(R(n_1, D)) \preceq_D^{v \cdot \sigma(f)} f(R(n_2, D))$$
where $\sigma(f) = +1$ for monotone-increasing functions (e.g., `MAX`, `SUM`, `COUNT`), and $\sigma(f) = -1$ for monotone-decreasing ones (e.g., `MIN`).
- **(Value → Value)**: If an expression or scalar operator op_v is transformed to another form with monotonic direction $\sigma(op_v)$, then the resulting value-level outputs satisfy:
$$V_{n_1}(g, c) \preceq_D^{v \cdot \sigma(op_v)} V_{n_2}(g, c)$$
This covers arithmetic transformations (`+`, `*2` with $c > 0$) and functional ones (`MAX`→`MIN`).
- **(Value → Set)**: If a value expression V_n feeds into a predicate or filtering operator op_s, and $V_{n_1} \preceq_D^v V_{n_2}$, then the induced output relations satisfy:
$$R(n_1, D) \preceq_D^{s \cdot \sigma(op_s)} R(n_2, D)$$
where $\sigma(op_s) = +1$ for monotone-increasing predicates (e.g., `x > c`, where larger values of `x` make the condition more likely to hold and thus expand the result set), and

$\sigma(op_s) = -1$ for monotone-decreasing ones (e.g., `x < c` or `NOT EXISTS`, where larger values of `x` make the condition less likely to hold, causing the result set to shrink).

Remark. In this definition, n_1 and n_2 are not restricted to complete queries. They can represent corresponding subqueries, expressions, or predicates within a single query or across two query variants. The relation $\leq_D^\alpha$ thus captures how a local semantic approximation propagates through SQL operators according to their monotonic behavior, bridging the value- and set-level semantics within the same unified framework.

Intuitively, the propagation mechanism provides the semantic bridge between *tuple-level inclusion* and *value-level monotonicity.* Set-level approximations can trigger value changes through monotone o perators, while value level changes can, in turn, alter the query result set when the affected values participate in predicates. This bidirectional propagation enables comprehensive reasoning over multi-layer SQL dependencies.

Example 3.3 (Set $\longrightarrow$ Value Propagation)**.** Consider two queries over a table $t_1(c_1, c_2)$:

q_1 : **SELECT MAX**(c_1) **FROM** t_1 **WHERE** $c_2 < 100$

q_2 : **SELECT MAX**(c_1) **FROM** t_1 **WHERE** $c_2 < 200$

In the query structure, let n_1 and n_2 denote the `WHERE` clause nodes of q_1 and q_2, respectively. The condition $c_2 < 100$ in n_1 is stricter than $c_2 < 200$ in n_2, so the rows selected by n_1 form a subset of those selected by n_2. The parent node of these filters is the aggregation operator `MAX`, which is monotone increasing: when more rows are included, the maximum value of c_1 can only increase or remain the same. As a result, the difference at the set level (fewer or more tuples) propagates upward to a difference at the value level (smaller or larger aggregated value).

Intuitively, n_1 and n_2 illustrate how a local change in the filter condition at the set level can influence the aggregated result value, demonstrating the propagation from Set to Value.

Example 3.4 (Value $\longrightarrow$ Set Propagation)**.** Consider two semantically related queries over a table $t_1(c_1)$:

q_1 : **SELECT** $*$ **FROM** (**SELECT MAX**(c_1) **AS** x **FROM** t_1) **AS** *subq* **WHERE** $x > 100$

q_2 : **SELECT** $*$ **FROM** (**SELECT MIN**(c_1) **AS** x **FROM** t_1) **AS** *subq* **WHERE** $x > 100$

The two queries differ only in the inner aggregation. Let n_1 and n_2 denote the aggregation nodes `MAX(c_1)` and `MIN(c_1)`, respectively. Changing `MAX` to `MIN` decreases the derived value x. Since the outer predicate `x > 100` is monotone increasing in x, smaller x values make the condition harder to satisfy, resulting in fewer output tuples. Consequently, the result of q_2 becomes a subset of q_1, showing a typical Value $\longrightarrow$ Set propagation.

These propagation behaviors connect the two approximation dimensions, allowing a single mutation at any AST node (e.g., `MAX`$\longrightarrow$`MIN`) to yield predictable, analyzable effects on both result structure and result values. The unified propagation model forms the semantic foundation of our testing framework.

4 Approach

4.1 Overview

We illustrate the overall workflow of our approach in Figure 1. The entire process follows a generate–mutate–verify paradigm designed to uncover logic bugs in DBMSs. In the pre-processing phase, we randomly populate multiple tables in a test database, following standard DBMS random testing practices [18]. This randomized setup provides a diverse and unbiased data distribution for subsequent query evaluations. Next, our system generates a syntactically valid SQL query that serves as the original query. It then parses the query and traverses its AST to identify which grammatical constructs can be safely mutated. Based on the SQL query approximation models defined in Section 3, the system automatically synthesizes several approximate queries by mutating the original queries. After the mutated queries are constructed, the framework performs approximation propagation analysis to reason how local semantic changes at the mutated node propagate through the SQL AST. This analysis establishes the global query-level approximation relation (either at the set level or the value level) between the original and mutated queries. Finally, both queries are executed on the tested DBMS instance, and their outputs are compared against the predicted approximation relation. Any violation of this expected relation indicates a potential logical inconsistency in the DBMS.

In the following sections, we introduce each core component in detail, including the construction of the original query (Section 4.2), the design of approximate mutators (Section 4.3), the propagation algorithm (Section 4.4), and the result checking procedure (Section 4.5).

4.2 Construction of the Original Query

To support diverse SQL structures in testing, our framework constructs original queries based on a compact yet expressive SQL grammar, as illustrated in Figure 2. This grammar extends the baseline adopted in Pinolo [8] by incorporating a richer set of syntactic components, enabling the generation of more varied queries. Compared with Pinolo, which mainly focuses on basic `SELECT-FROM-WHERE` statements, our implementation supports a broader range of SQL features, including comprehensive `JOIN` types and conditions, `WITH`-clause based common table expressions (CTEs), and row-level locking modes such as `FOR UPDATE`. It also introduces extended functionalities such as `GROUP BY` with `HAVING` filters and `ORDER BY` clauses, nested function calls

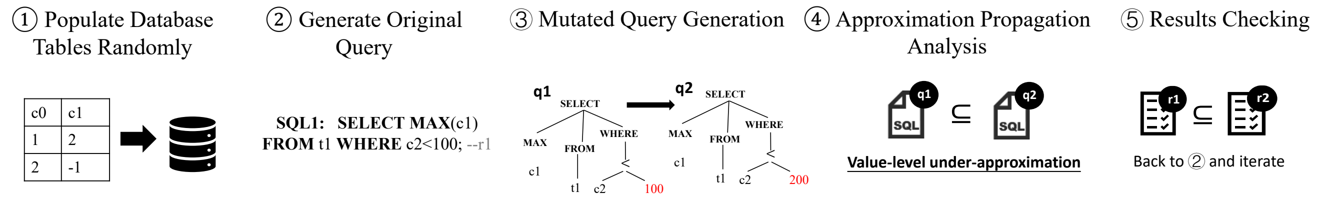

① Populate Database Tables Randomly
② Generate Original Query
③ Mutated Query Generation
④ Approximation Propagation Analysis
⑤ Results Checking
c0
c1
1
2
2
-1
SQL1: SELECT MAX(c1) FROM t1 WHERE c2<100; --r1
q1
SELECT
MAX
c1
FROM
t1
WHERE
<
c2
100
q2
SELECT
MAX
c1
FROM
t1
WHERE
<
c2
200
q1
SQL
⊆
q2
SQL
Value-level under-approximation
r1
⊆
r2
Back to ② and iterate

Figure 1. Overview of our approach.

Statement	S	::=	Q | Q FOR LM | WITH WI (, WI)* Q [FOR LM]
Query	Q	::=	QC [OB] [L] | Q SO Q
SelectCore	QC	::=	SELECT [DISTINCT] SL FC [WH] [GB]
SelectList	SL	::=	VE [AS id] (, VE [AS id])*
FromClause	FC	::=	FROM TR (, TR)*
TableRef	TR	::=	id [AS id] | (Q) AS id | TR $JTYP$ JOIN TR [ON BE]
JoinType	$JTYP$	::=	INNER | LEFT OUTER | RIGHT OUTER | CROSS
Where	WH	::=	WHERE BE
GroupBy	GB	::=	GROUP BY CR (, CR)* [HAVING BE]
OrderBy	OB	::=	ORDER BY VE [ASC | DESC] (, VE [ASC | DESC])*
Limit	L	::=	LIMIT int
SetOp	SO	::=	UNION | UNION ALL | INTERSECT | EXCEPT
BoolExpr	BE	::=	VE COP VE | NOT BE | BE AND BE | BE OR BE
		|	EXISTS (Q) | VE IS [NOT] NULL
		|	VE [NOT] IN (Q | VE (, VE)*)
CompOp	COP	::=	= | <> | < | > | <= | >= | LIKE | BETWEEN
ValueExpr	VE	::=	CASE WHEN BE THEN VE ELSE VE END | (Q)
		|	| id | $FCALL$ | $CONST$
FuncCall	$FCALL$	::=	id ([VE (, VE)*])
LockMode	LM	::=	UPDATE | SHARE | NO KEY UPDATE | KEY SHARE
WithItem	WI	::=	id AS (Q)
ColumnRef	CR	::=	id [. id]
Const	$CONST$	::=	integer, float, string, date, time, timestamp, binary, TRUE, FALSE, NULL

Figure 2. Compact SQL syntax supported by our approach.

with type casting and CASE expressions. These extensions collectively allow our system to generate queries that better reflect the syntactic and semantic richness of real-world DBMS workloads.

A key difference between our design and Pinolo lies in the treatment of aggregation and grouping operations. Pinolo does not support GROUP BY or HAVING clauses, as aggregation and window functions disrupt the set-containment relation on which its approximation model relies. Specifically, once aggregation is introduced, the query output no longer preserves monotonic inclusion over tuples—adding or removing rows may alter the aggregated values in non-monotonic ways. By contrast, our approach introduces a new class of value-semantic approximation relations as discussed in Section 3, that model how numerical and aggregated results change under local semantic perturbations.

4.3 Approximate Mutators

4.4 Approximation Propagation Analysis

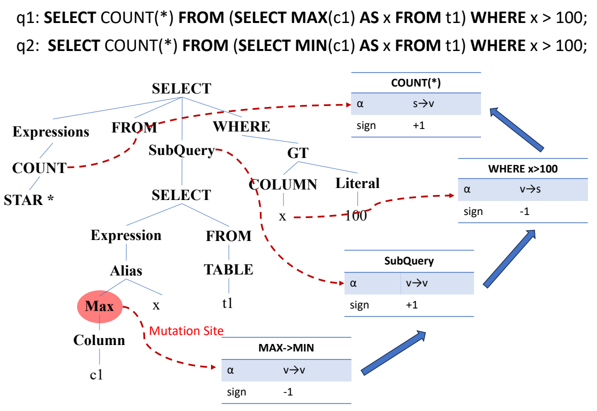
In this section, we further develop an executable algorithm to determine how local semantic changes propagate through the SQL AST. The main purpose of this algorithm is to formalize the top-down reasoning process introduced in Definition 3.3 into a systematic, bottom-up propagation procedure that connects local node mutations with their global semantic consequences at the query level.

Algorithm 1 presents the overall propagation process. Initially (Line 1–3), the algorithm receives the mutated node information $node\_info$ and the complete AST of the query. It first identifies the mutated node n_{mut} and constructs its ancestor chain from the mutation site to the query root, represented as list $= [n_{mut}, \ldots, n_{root}]$. This structure enables the algorithm to traverse each parent operator sequentially and reason about how the mutation propagates upward. The algorithm then initializes the local semantic level of the mutation (Line 5–7): if the mutated node involves predicates or subqueries, it starts at the set level ($\alpha(n_{mut}) = s$); otherwise, for expressions or aggregations, it starts at the value level ($\alpha(n_{mut}) = v$). A direction accumulator *sign* is also initialized to +1 to indicate that propagation initially preserves directionality. In the propagation stage (Line 9–17), the algorithm iteratively traverses each parent node of n_{mut} in a bottom-up manner. Before propagating, the algorithm invokes DependsOn(n_i, n_{i-1}) to check whether the parent node semantically depends on the mutated child node—by referencing its output column (for value-level nodes) or embedding it as a subquery or relation (for set-level nodes). Only dependent nodes are considered for propagation. For each parent operator op_i (Line 9), it consults Table 1 to retrieve the corresponding semantic mapping ($\alpha_{in} \rightarrow \alpha_{out}$) and its monotonic direction $\sigma(op_i) \in \{+1, -1\}$ (Line 13). The semantic level α is updated according to the operator's input-output mapping (e.g., Set⟶Value for aggregation or Value⟶Set for predicate filters), while the cumulative direction *sign* is updated multiplicatively ($sign \leftarrow sign \cdot \sigma(op_i)$), preserving or reversing the relation based on operator polarity (Line 14–16). This recursive process effectively tracks the path of semantic transformation from the mutation site to the query output. Finally, at the root node (Line 19–25), the algorithm derives the final query-level relation—returning either a set-level ($R(q_1, D)! \preceq^s_D !R(q_2, D)$) or value-level ($V_{q_1}(g, c)! \preceq^v_D !V_{q_2}(g, c)$) approximation, determined by the accumulated propagation direction.

Algorithm 1 Approximation Propagation across SQL AST

```
Input: Mutated node info $node\_info$, SQL AST $AST$
Output: Final query-level approximation between original and mutated queries
// Step 1: Initialization
Identify mutated node $n_{mut}$.
Build ancestor chain list = $[n_{mut}, \ldots, n_{root}]$.
// Step 2: Local relation at mutation site
Decide initial level $\alpha(n_{mut}) \in \{s, v\}$ by node type.
Set local relation $\mathcal{R}(n_{mut}) \leftarrow (\preceq^{\alpha(n_{mut})})$.
Initialize direction accumulator $sign \leftarrow +1$.
// Step 3: Bottom-up propagation
for each parent node $n_i$ in $list$ (from child to root) do
    Determine operator type $op_i$ at $n_i$.
    if not DependsOn($n_i, n_{i-1}$) continue
    end if
    Lookup rule of $op_i$ in Table 1 to get $(\alpha_{in} \to \alpha_{out}, \sigma(op_i))$.
    Update level: $\alpha(n_{i+1}) \leftarrow \alpha_{out}$.
    Update direction: $sign \leftarrow sign \cdot \sigma(op_i)$.
    Propagate symbolically: $\mathcal{R}(n_{i+1}) \leftarrow \mathcal{R}(n_i)$ with $sign$ applied.
end for
// Step 4: Materialize root-level relation
if $\alpha(n_{root}) = s$
    return $R(q_1, D) \preceq^s_D R(q_2, D)$ with sign +,
    or $R(q_1, D) \succeq^s_D R(q_2, D)$ with sign −.
else
    return $V_{q_1}(g, c) \preceq^v_D V_{q_2}(g, c)$ with sign +,
    or $V_{q_1}(g, c) \succeq^v_D V_{q_2}(g, c)$ with sign −.
end if
function DependsOn(parent, child)
    if $\alpha(child) = v$
        return parent.exprs contains column or alias from child
    else if $\alpha(child) = s$
        return parent.references(child) as subquery or table
    else
        return false
    end if
end function
```

As illustrated in Figure 3, the mutation occurs at the aggregation node where `MAX(c1)` is replaced by `MIN(c1)`, initializing a value-level relation ($\alpha = v$) with reversed monotonicity ($\sigma = -1$). Following Algorithm 1, the algorithm then traverses its ancestor chain list = $[n_{\text{MAX}}, n_{\text{subq}}, n_{\text{WHERE}}, n_{\text{COUNT}}]$ to evaluate how this local change propagates upward through the query structure. When the propagation reaches the subquery node (`SELECT ... AS x`), the semantics are lifted from the value level to the set level ($\alpha : v \to s$, $\sigma = +1$). According to Table 1, a subquery appearing in the `FROM` clause acts as a data source that preserves tuple inclusion—expanding the underlying relation results in a superset of tuples, thus maintaining positive monotonicity. The subsequent `WHERE x>100` clause applies a filtering operator that preserves the set level but reverses direction ($\alpha : s \to s$, $\sigma = -1$). Finally, the outer `COUNT(*)` operator aggregates the resulting set back to a



```
q1: SELECT COUNT(*) FROM (SELECT MAX(c1) AS x FROM t1) WHERE x > 100;
q2:  SELECT COUNT(*) FROM (SELECT MIN(c1) AS x FROM t1) WHERE x > 100;
```

Figure 3. Example of how the proposed algorithm propagates local semantic changes across the SQL AST.

value-level ($\alpha : s \to v$, $\sigma = +1$). The cumulative sign is therefore $(-1)\times(+1)\times(-1)\times(+1) = +1$, yielding a final value-level approximation $V_{q_1}(g, c) \preceq^v_D V_{q_2}(g, c)$, which indicates that the output of q_1 forms a value-level under-approximation of q_2.

4.5 Results Checking

The primary purpose of results checking is to verify whether the outputs of the original and mutated SQL queries align with the predicted approximation relations, as defined in the previous section. Unlike traditional methods that rely on strict equivalence or set inclusion checks, which often miss subtle semantic differences, our approach incorporates both set-semantic and value-semantic approximation relations. This allows us to detect a wider range of logical inconsistencies in DBMS query results. In our approach, after executing both queries, we compare their results based on the expected approximation relation (set or value). Specifically, for set-semantic approximation, we check if the result set of the mutated query is a subset or superset of the original query's result set, as predicted by the approximation relation. For value-semantic approximation, we compare the values in the target columns of both queries, ensuring that they satisfy the expected monotonic relationship (either increasing or decreasing). A violation of this relation signals a logical inconsistency in the DBMS's query processing.

5 Implementation

We implemented our approach ValScope as a DBMS testing system, which was written in Python with 8,055 lines of code. The source code of our tool is hosted in the github repository. ✎ [lin: add a url] The following describes the important implementation details.

Database Population. We adopt the approach Pinolo [8] to generate random database instances from scratch, while also covering most of the data types as specified in the SQL

Table 1. Operator Types and Semantic Propagation Rules.

Operator Type	Semantic Mapping	Monotonic Direction $\sigma(op)$	Example and Interpretation
Aggregation Functions	Set → Value	+1: `MAX`, `SUM`, `COUNT` −1: `MIN`	Expanding the input set increases the aggregated result for `MAX`/`SUM`/`COUNT`, or decreases it for `MIN`.
Predicate / Filter	Value → Set	+1: $x > c, x \geq c$ −1: $x < c, x \leq c$	Increasing the compared value makes $x > c$ easier (result set expands) and $x < c$ harder (result set shrinks).
Logical Operator	Set → Set	+1: `EXISTS`, `OR`, `AND` −1: `NOT`, `NOT EXISTS`	`OR`/`EXISTS`/`AND` are monotone increasing w.r.t. each input (union/exists/intersection); `NOT`/`NOT EXISTS` flip inclusion.
Arithmetic / Expression	Value → Value	Depends on operator sign	Linear arithmetic preserves or reverses direction: `x + k` (+1 if $k > 0$), `x * k` (+1 if $k > 0$, −1 if $k < 0$), and `-x` flips monotonicity.
Join / Projection / Set Op	Set → Set	+1: `JOIN`, `SELECT`, `UNION`, `INTERSECT` mixed: `EXCEPT` (*left* +1 / *right* −1)	Structural operators propagate tuple inclusion: `JOIN`, basic `SELECT`, `UNION`, `INTERSECT` are monotone increasing; `EXCEPT` increases with the left input but decreases with the right input.
Subquery / CTE	Context-dependent	+1: as `FROM`/CTE source inherits parent: scalar subquery	Monotonicity depends on usage context: (1) In the `FROM` clause (e.g., `SELECT * FROM (SELECT a FROM t)`), it acts as a data source and preserves inclusion (+1). (2) As a scalar subquery (e.g., `WHERE x > (SELECT AVG(y) FROM t)`), it inherits the outer predicate's direction ($> \Rightarrow +1, < \Rightarrow -1$). (3) For a CTE (e.g., `WITH cte AS (...) SELECT * FROM cte`), it behaves similarly to a `FROM` subquery, typically monotonic (+1).

documentation. Unlike existing methods of random database instance generation, ValScope also takes into account type-specific behaviors and boundary cases. For each column, values are generated according to the constraints of the respective data type to ensure validity. This helps prevent common runtime errors such as INSERT failures caused by out-of-range or incorrectly formatted values. To increase the likelihood of triggering type-related errors, we also introduce boundary values, including extremely large or small numbers, as well as text with mixed case. Additionally, to test specific arithmetic operations and function-related mutation operators, we explicitly specify the sign (positive or negative) for certain columns.

Test Case Generation and Parsing. Similar to SQL generators like SQLSmith [1] and Go-randgen [15], we implement ValScope from scratch to generate original queries. ValScope ensures the consistency of data dependencies, function dependencies, and expression dependencies in query generation. It carefully selects tables and columns based on their usage patterns, applies the correct functions with compatible argument types, and maintains type compatibility across operations, ensuring semantic correctness. This allows ValScope to achieve higher semantic accuracy than existing generators such as SQLSmith and Go-randgen, particularly in handling complex queries with multiple nested subqueries. To apply the approximate mutators to original queries, we use SQLGlot [14], which accepts the same context-free grammar used in seed query generation, to generate ASTs of original queries for mutation.

Bug Report Post-processing. ValScope evaluates queries in the tested DBMS to obtain query results. It is important to note that inconsistent query results often arise during our testing process. To avoid repetitive bug reports and make the test results easier to understand, we first use SQLess [12] to simplify the bugs and pinpoint their root causes. Additionally, we follow the approach from Pinolo [8] to remove duplicates.

Supported DBMSs and Adaptation. ValScope is primarily designed around the MySQL syntax, ensuring robust support for MySQL-based databases, such as TiDB and OceanBase. The architecture of ValScope is modular and decoupled, which facilitates easy adaptation to different DBMS dialects. To integrate support for additional DBMS dialects, users only need to modify the relevant components to accommodate the specific syntax, data types, and operations of the target DBMS. This adaptation is similar to extending SQLSmith [1] and can be achieved with just a few hundred lines of code. A more streamlined approach is to leverage the latest extension tool, QTRAN [13], which automates the process of adapting metamorphic-oracle-based logical bug detection techniques for multi-DBMS dialect support.

6 Evaluation

6.1 Experimental Setup

6.2 Bug Detection

6.3 Bug Diversity

6.4 Comparative Study

7 Related Work

Logical Bug Detection in DBMSs. A variety of approaches have been proposed to detect logical bugs in DBMSs [6, 8, 10, 16–18, 20–22]. NoREC [16] transforms an optimizable query into a non-optimizable form and detects semantic inconsistencies by comparing their outputs. TLP [17] divides query predicates into multiple subqueries and verifies that the union of their results is equivalent to the original query. PQS [18] constructs queries expected to retrieve a specific pivot row, while DQE [20] detects logical bugs by comparing whether different SQL queries with the same predicate access the same rows in the database. TQS [22] decomposes wide tables into smaller ones and uses the base table as ground truth for correctness validation. EET [10] applies expression-preserving transformations, and RADAR [21] compares query results between databases with and without metadata to identify semantic flaws. EDC [6] detects logical bugs by substituting expressions with precomputed equivalent data and checking for inconsistent results, while CODDTEST [23] leverages constant folding and propagation to generate equivalent queries.

Most of these methods rely on constructing **equivalent SQL pairs**, detecting bugs when such equivalence fails to hold. Recently, PINOLO [8] generalizes this paradigm through set-level approximation, where predicates are relaxed or restricted to generate over- and under-approximate queries, and correctness is judged by inclusion or containment relations between result sets. Building on this foundation, our work extends the set semantics to the value semantics level, enabling the detection of subtler logical errors that cannot be captured by set inclusion alone. By combining set-level consistency with value-directional reasoning, our approach establishes a multi-dimensional criterion for identifying query approximations and uncovering deeply hidden semantic faults.

DBMS Test-Case Generation. A variety of approaches have been proposed for generating diverse test cases for DBMSs, with the aim of improving coverage and revealing potential bugs. These techniques [1, 7, 9, 24] typically focus on generating valid and varied SQL queries, but may not specifically target logical bugs. SQLSMITH [1] is a grammar-based DBMS fuzzer that embeds SQL grammar rules to generate complex SQL queries. It uses a random walk approach to explore the SQL syntax and generate a wide range of queries. SQUIRREL [24] is a mutation-based DBMS fuzzer which introduces an intermediate representation for SQL queries and models dependencies between SQL statements, enabling the generation of queries that contain multiple SQL operations. GRIFFIN [7] uses a grammar-free mutation approach, where it mutates SQL queries based on DBMS state information encapsulated in a metadata graph. QTRAN [13] is a LLM-based approach that can automatically translate test cases from other DBMSs. DYNSQL [9] takes a dynamic approach by interacting with the DBMS to capture the latest state information, allowing for the incremental generation of valid and complex queries. These techniques aim to prevent semantic errors and improve the diversity of generated queries. In addition to general query generation, some approaches have been specifically designed to aid in bug detection. SQLRIGHT [11] leverages code coverage feedback to enhance test-case generation. This feedback provides insights into which parts of the DBMS code are exercised, increasing the chances of uncovering logical bugs in infrequently executed paths. QPG [2] takes a different approach by recording the query plans covered during DBMS testing and prioritizing the mutation of queries that trigger new query plans. This targeted mutation is more likely to expose logical bugs that might otherwise remain undetected.

These general query generation approaches complement VALSCOPE. While approaches like SQLSMITH, SQUIRREL, and GRIFFIN focus on generating diverse queries, VALSCOPE can help identify logical bugs hidden in complex or rarely executed query logic. Conversely, the test cases generated by these methods can provide VALSCOPE with high-quality and varied queries, expanding its ability to uncover bugs. Together, these approaches offer a comprehensive strategy for DBMS testing, improving both query coverage and the detection of logical bugs.

8 Conclusion

References

[1] Anse1. n.d.. SQLsmith: A Random SQL Query Generator. https://github.com/anse1/sqlsmith. Accessed: 2025-04-30.

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

[3] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. 2007. A genetic approach for random testing of database systems. In *Proceedings of the 33rd international conference on Very large data bases*. 1243–1251.

[4] Donald D Chamberlin and Raymond F Boyce. 1974. SEQUEL: A structured English query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. 249–264.

[5] Edgar F Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387.

[6] WENQIAN DENG, JIE LIANG, ZHIYONG WU, JINGZHOU FU, and YU JIANG. 2025. Detecting Logic Bugs in DBMSs via Equivalent Data Construction. (2025).

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

[8] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting logical bugs in database management systems with approximate query synthesis. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 345–358.

[9] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. {DynSQL}: Stateful fuzzing for database management systems with complex and valid {SQL} query generation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4949–4965.

[10] 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)*. 821–835.

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

[12] Li Lin, Zongyin Hao, Chengpeng Wang, Zhuangda Wang, Rongxin Wu, and Gang Fan. 2024. SQLess: Dialect-Agnostic SQL Query Simplification. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 743–754.

[13] Li Lin, Qinglin Zhu, Hongqiao Chen, Zhuangda Wang, Rongxin Wu, and Xiaoheng Xie. 2025. QTRAN: Extending Metamorphic-Oracle Based Logical Bug Detection Techniques for Multiple-DBMS Dialect Support. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 731–752.

[14] Toby Mao. 2024. SQLGlot: A no-dependency SQL parser, transpiler, optimizer, and engine. https://github.com/tobymao/sqlglot Accessed: November 2024.

[15] PingCap. 2022. go-randgen. https://github.com/pingcap/go-randgen Online; accessed Dec 2025.

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

[17] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[18] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 667–682.

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

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

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

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

[23] Chi Zhang and Manuel Rigger. 2025. Constant Optimization Driven Database System Testing. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–24.

[24] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970.