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

Abstract

Pending...

1 Introduction

2 Background

3 SQL Query Approximation

A fundamental challenge in DBMS testing lies in the test oracle problem: given two semantically related SQL queries, it is often unclear whether their outputs should be identical or follow a predictable metamorphic relationship. 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 [1]. 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:

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

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

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

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:

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

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:

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

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. ✎ [lin: Check why can coexist?] In practice, the two forms often coexist:

$$q_1 \preceq_D^s q_2 \ \wedge\ q_1 \preceq_D^v q_2$$

which indicates that q_1 returns a subset of q_2's tuples and the corresponding values in the target columns are not larger.

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

$$t_1 = \begin{array}{c|c} c_2 & c_1 \\ \hline A & 10 \\ A & 20 \\ B & 5 \\ B & 7 \end{array}$$

Let the following two queries be defined:

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

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

The results are:

$$R(q_1, D) = \begin{array}{c|c} c_2 & \text{MAX}(c_1) \\ \hline A & 20 \\ B & 7 \end{array} \qquad R(q_2, D) = \begin{array}{c|c} c_2 & \text{MIN}(c_1) \\ \hline A & 10 \\ B & 5 \end{array}$$

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

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

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* ($\alpha_{in} \to \alpha_{out}$), 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 $\preceq_D^{\alpha}$ 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 operators, 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 $\rightarrow$ Value Propagation)**.** Consider two queries over a table $t_1(c_1, c_2)$:

$$q_1 : \textbf{SELECT MAX}(c_1)\ \textbf{FROM}\ t_1\ \textbf{WHERE}\ c_2 < 100$$

$$q_2 : \textbf{SELECT MAX}(c_1)\ \textbf{FROM}\ t_1\ \textbf{WHERE}\ c_2 < 200$$

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 $\rightarrow$ Set Propagation)**.** Consider two semantically related queries over a table $t_1(c_1)$:

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

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

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 $\rightarrow$ Set propagation.

These propagation behaviors connect the two approximation dimensions, allowing a single mutation at any AST node (e.g., `MAX`→`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 [2]. 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

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

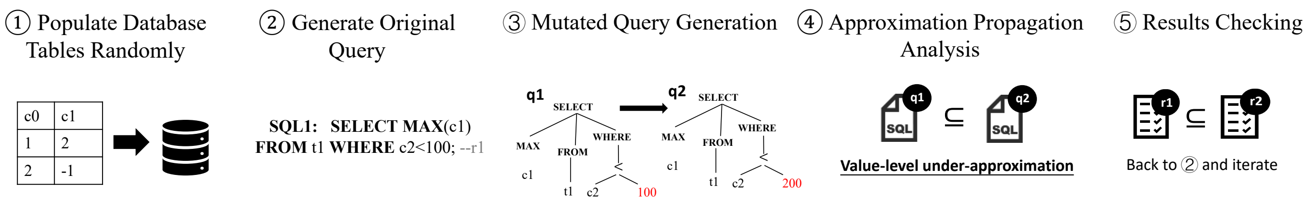

Figure 1. Overview of our approach.

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)
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, string, float, TRUE, FALSE, NULL

Figure 2. Compact SQL syntax supported by our approach.

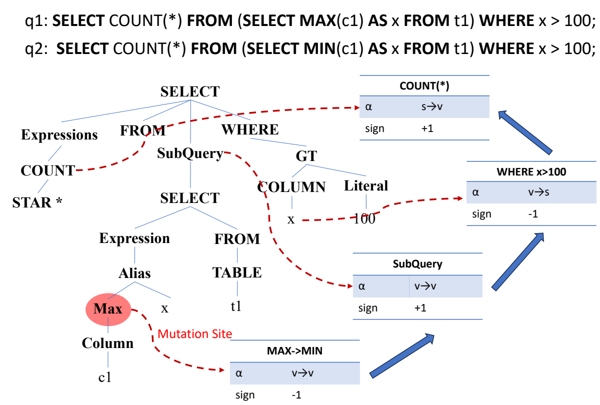

Figure 3. Example of how the proposed algorithm propagates local semantic changes across the SQL AST.

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.

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

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 $(\alpha_{in} \rightarrow \alpha_{out}, \sigma(op_i))$.
    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

the value level to the set level ($\alpha : v \rightarrow s$, $\sigma = +1$). 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 ($\alpha : s \rightarrow s$, $\sigma = -1$). Finally, the outer `COUNT(*)` operator aggregates the resulting set back to a value-level ($\alpha : s \rightarrow v$, $\sigma = +1$). 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

5 Implementation

References

[1] 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.

[2] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. 2023. Testing graph database engines via query partitioning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 140–149.

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).