Finding Missed Optimizations in DBMSs through Unbalanced Short-Circuit Query Construction

JINHUI LAI, Nanchang University, China
CHI ZHANG, Tsinghua University, China
JIE LIANG, Beihang University, China
ZIHAO ZENG, Nanchang University, China
ZHIYONG WU, Tsinghua University, China
JINGZHOU FU, Tsinghua University, China
CHIJIN ZHOU, East China Normal University, China
SHUAI MA, Beihang University, China
YU JIANG, Tsinghua University, China
ZICHEN XU*, Nanchang University, China

DBMSs underpin modern data-intensive applications, where performance directly impacts system responsiveness and user experience. To improve efficiency, DBMSs employ many sophisticated optimization techniques for SQL queries. However, the complexity of SQL queries and DBMS architectures often causes implementations to miss potential optimization opportunities, resulting in performance degradation, inefficient resource utilization, and diminished user experience. Identifying these missed optimizations is challenging due to the intricate interactions among query semantics, optimizer decisions, and execution behaviors. Existing approaches have detected many performance issues when implemented optimizations perform poorly. However, they rarely reveal optimizations that were entirely missed, leaving many performance gaps unaddressed.

In this paper, we present SCor, a black-box approach for identifying missed optimizations in DBMSs through unbalanced short-circuit query construction. Our key insight is that the results of many queries can be determined without full execution, yet DBMSs still execute the entire query, revealing missed optimizations. SCor realizes it by constructing unbalanced short-circuit queries, where the result can be obtained from low-cost operations alone. If the DBMS still executes the full query with high-cost operations, it indicates missed optimizations. Since short-circuit patterns are prevalent in SQL queries and can be flexibly embedded into diverse query structures, SCor can be systematically applied to expose missed optimizations across a wide range of queries. Our evaluation of SCor across 11 widely-used DBMSs, revealed 153 previously undetected performance bugs resulting from missed optimizations, including 2 bugs in Oracle and 3 bugs in PostgreSQL. Among all reported bugs, 125 have been confirmed by developers, and 33 have already been fixed.

CCS Concepts: • **Information systems** → **Query optimization**.

Additional Key Words and Phrases: DBMS Testing, Missed Optimizations, Short-Circuit Evaluation

*Zichen Xu is the corresponding author.

Authors' Contact Information: Jinhui Lai, jinhuilai@email.ncu.edu.cn, Nanchang University, China; Chi Zhang, chizhang@mail.tsinghua.edu.cn, Tsinghua University, China; Jie Liang, liangjie.mailbox.cn@gmail.com, Beihang University, China; Zihao Zeng, zihaozeng0021@gmail.com, Nanchang University, China; Zhiyong Wu, 253540651@qq.com, Tsinghua University, China; Jingzhou Fu, fuboat@outlook.com, Tsinghua University, China; Chijin Zhou, tlock.chijin@gmail.com, East China Normal University, China; Shuai Ma, mashuai@buaa.edu.cn, Beihang University, China; Yu Jiang, jiangyu198964@126.com, Tsinghua University, China; Zichen Xu, xuz@ncu.edu.cn, Nanchang University, China.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2836-6573/2026/6-ART184
https://doi.org/10.1145/3802061

ACM Reference Format:
Jinhui Lai, Chi Zhang, Jie Liang, Zihao Zeng, Zhiyong Wu, Jingzhou Fu, Chijin Zhou, Shuai Ma, Yu Jiang, and Zichen Xu. 2026. Finding Missed Optimizations in DBMSs through Unbalanced Short-Circuit Query Construction. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 184 (June 2026), 24 pages. https://doi.org/10.1145/3802061

1 Introduction

Database management systems (DBMSs) are a critical foundation of modern data-intensive applications, where their performance directly affects system responsiveness and user satisfaction. Over the past decades, both DBMS vendors and researchers have devoted tremendous efforts to improving query performance, leading to the development of many sophisticated optimization techniques for SQL execution [20]. Despite these advances, DBMS still suffers from performance bugs, where certain queries remain insufficiently optimized and exhibit unexpected slowdowns. In this work, we refer to such cases as **missed optimizations**, which occur when the optimizer either fails to select the most efficient execution plan among available alternatives or does not implement a more effective optimization strategy.

The prevalence of missed optimizations can be attributed to the inherent complexity of SQL semantics [4] and architectures of DBMSs [11], which together often lead to incomplete or suboptimal optimization decisions. For example, Figure 1 illustrates a case of a missed optimization we identified in Oracle 23ai, which resulted in a 2995× performance slowdown. The two queries in this case are semantically equivalent and should both return immediately, since the `OR` expression contains a constant `TRUE` operand that makes the subquery evaluation unnecessary. However, while the first query is efficiently optimized through short-circuit evaluation, the other one suffers from a missed optimization because the optimizer fails to reorder the operands [16, 20] and still executes the expensive subquery first. As a result, the query scans one billion rows in the table `t1` and takes 29.95 seconds to complete. The example demonstrates that even mature commercial systems can overlook fundamental optimization opportunities, leading to dramatic performance degradations.

However, detecting such missed optimizations in DBMSs remains a non-trivial task due to several inherent challenges: (1) **The Vast and Intricate Optimization Space of SQL.** SQL language itself is highly expressive, encompassing diverse constructs such as nested queries, complex predicates, and rich semantic variations. This complexity is further exacerbated by proprietary extensions introduced by different DBMS vendors, which make the optimization space even harder to reason about. (2) **The Opacity of Internal Optimization Mechanisms.** The internal optimization strategies adopted by commercial DBMSs (e.g., Oracle [34]) are typically undisclosed. As a result, it is difficult to determine which optimizations are actually applied during query execution or whether certain optimization opportunities were intentionally omitted. (3) **The Absence of a Reliable Performance Oracle.** DBMS performance testing lacks a reliable oracle that specifies the reasonable execution time for a given query Q and database D. Without such ground truth, it is challenging to distinguish a genuine missed optimization from an inherently expensive query.

Some automated testing approaches have been widely adopted to detect performance issues for DBMSs [1, 18, 23, 24, 54, 55]. HULK [55] and APOLLO [18] detect performance regressions by comparing query execution across different systems or versions. MOZI [23] and PUPPY [54] vary optimizer configurations to trigger different execution plans and reveal performance degradation. CERT [1] focuses on cardinality estimation by deriving restrictive queries to identify estimation errors that may affect optimization. While effective in uncovering issues in implemented optimizations, none of these approaches explicitly targets missed optimization opportunities, where the optimizer fails to choose an execution strategy that would yield more efficient query evaluation.

1. Generate table t1 and insert t1 with one billion rows

```
CREATE TABLE t1(c1 NUMBER(10));
INSERT INTO t1 SELECT LEVEL FROM dual CONNECT BY LEVEL <= 1000000000;
```

2. Generate two queries and execute them

```
-- Positive case: evaluate the cheaper operand (a constant TRUE) first
SELECT TRUE OR (SELECT MIN(t1.c1) FROM t1)>0 ; -- 0.01s
```

```
-- Negative case: evaluate the expensive operand (a subquery) first
SELECT (SELECT MIN(t1.c1) FROM t1)>0 OR TRUE ; -- 29.95s
```

Fig. 1. A missed optimization detected by SCor in Oracle 23ai that leads to a 2995× performance degradation.

In this paper, we propose SCor, an automated black-box testing approach for detecting missed optimization opportunities in DBMSs. Our key insight is that *if a query's result can be determined from a partial execution, yet the DBMS still executes the entire query, it indicates a missed optimization.* We realize this insight using short-circuit evaluation [35, 36, 38], where in a branched expression (e.g., `OR` or `AND`), the second branch is executed only if the first branch cannot determine the overall result. Specifically, we summarize five categories comprising ten representative short-circuit query patterns commonly found in official documentation [8, 26, 29, 35, 36, 40] of popular DBMSs. These patterns enable testing both short-circuit evaluation and related optimizations, such as cost-based operand prioritization. Based on these patterns, SCor generates unbalanced queries, in which the expressions that determine the query result are low-cost, while the expressions that are unnecessary for computing the result are high-cost. For instance, the simplified case in Figure 1 also demonstrates how we leverage unbalanced short-circuit queries to identify a missed optimization. In pattern p `OR` *trueExpr*, *trueExpr* can be instantiated as the constant `TRUE`, while p can be instantiated as an expensive subquery, such as `(SELECT MIN(c1) FROM t1) > 0` over a large table `t1`. Comparing its execution time against a baseline query using a small table `t0` reveals whether the DBMS executes the full query unnecessarily. A substantial performance difference between these queries serves as empirical evidence of a potential missed optimization opportunity.

We implemented SCor as a DBMS testing framework and applied it to 11 widely-used, mature DBMSs. Using this approach, we identified a total of 153 previously unknown performance bugs caused by missed optimizations across these systems, including 2 in Oracle [34], 3 in PostgreSQL [39], 13 in MySQL [28], 21 in MariaDB [25], 6 in CockroachDB [7], 12 in ClickHouse [6], 26 in TiDB [50], 9 in OpenGauss [33], 25 in OceanBase [31], 21 in SQLite [47], and 15 in DuckDB [12]. Of them, 125 have been confirmed, 33 have been fixed, 28 are being investigated. Many of these missed optimizations have persisted for extended periods, with the longest remaining undetected for 25.1 years. Moreover, DBMS vendors consider such missed optimizations highly important for system performance. For example, the OceanBase developer confirmed our report, stating: "*This issue will be handled as a requirement and optimized in future versions.*" These results demonstrate the effectiveness of SCor in detecting long-latent, non-trivial missed optimizations in DBMSs. Overall, we make the following contributions:

- We find that missed optimizations are widespread in DBMSs and highly detrimental to query performance, yet remain difficult for existing approaches to detect.
- We propose SCor, an automated black-box approach to detect performance issues caused by missed optimizations in DBMSs. It leverages ten representative patterns to generate queries that incorporate short-circuit evaluation, enabling the detection of missed optimizations.
- We implemented and evaluated the SCor on 11 widely-used DBMSs, uncovering 153 previously unknown missed optimizations, some of which have persisted for decades. Of them, 125 have been confirmed, 33 have been fixed.

2 Background

Query Optimization and Missed Optimizations. Query optimization [16, 20] is a fundamental process in DBMSs that determines the most efficient execution plan for parsed SQL queries. This task is inherently challenging due to SQL's declarative nature, which separates a query's logical structure from its physical execution. Modern optimizers address this complexity using cost-based models [16, 20] that evaluate alternative execution plans and often reorder operations to minimize estimated costs. For instance, a predicate written as $expr_1$ `OR` $expr_2$ may be executed as $expr_2$ `OR` $expr_1$ depending on cost estimates, illustrating how optimization decisions dictate the actual execution steps of a query. Despite these sophisticated optimization techniques, DBMSs can still fail to select the most efficient execution plan for certain queries. Such missed optimizations occur when the optimizer does not apply the most cost-effective execution strategy, resulting in unnecessary computation, degraded performance, and inefficient resource utilization. Detecting these missed optimizations remains challenging, as existing approaches [1, 18, 23, 24, 54, 55] typically observe performance anomalies without revealing where the optimizer fails.

Short-Circuit Evaluation. Short-circuit evaluation in DBMSs halts query execution once the final result is determined. For example, in $expr_1$ `OR` $expr_2$, if $expr_1$ evaluates to `TRUE`, $expr_2$'s execution becomes unnecessary. DBMSs primarily implement three types of short-circuiting: (1) Compile-time short-circuit, where syntactic constructs (e.g., `WHERE FALSE` [36] or `LIMIT 0` [29]) in a query enable the optimizer to infer a constant result or prune entire execution branches during query compilation. (2) Runtime short-circuit, where execution operators (e.g., semi-join [26] or `LIMIT n`) can stop processing once a termination condition is met, with the decision made dynamically during execution. (3) Statistics-aware short-circuit, where the optimizer leverages metadata such as table emptiness, selectivity estimates, or design choices like plan cache [27] to choose plans that are expected to exploit short-circuiting opportunities at runtime.

Metamorphic testing. A reliable test oracle is essential for identifying missed optimizations in DBMSs, as it defines the expected performance of a query and allows distinguishing genuine optimization misses from inherently expensive queries. Metamorphic testing [1] addresses this by constructing oracles through relations between inputs and outputs. For a given input I producing output O, the method derives a transformed input I' (with output O') and verifies whether a predefined relation holds between O and O'. SCor adapts this approach with the key insight that *if a query's result can be determined from partial execution, yet the DBMS executes the query entirely, this indicates a missed optimization.* We leverage short-circuit evaluation principles to construct such metamorphic relations. For example, given Q_1: `SELECT (SELECT MIN(t1.c1) FROM t1) > 0 OR TRUE` where `t1` is a large table, we create Q_2 by replacing `t1` with a small table or empty table `t0`, yielding `SELECT (SELECT MIN(t0.c1) FROM t0) > 0 OR TRUE`. Both queries should execute similarly fast since the `TRUE` operand makes the subquery unnecessary. A significant performance difference between Q_1 and Q_2 indicates a potential missed optimization opportunity.

3 Design of SCor

As Figure 2 shows, SCor operates in three steps. In step ①, SCor constructs unbalanced short-circuit query skeletons based on 10 representative patterns widely used in DBMSs [8, 26, 29, 35, 36, 40]. These skeletons retain SQL operational keywords while replacing specific elements (e.g., tables, columns, constants) with placeholders classified as low-cost or high-cost. Transformations such as subtree swapping are applied to increase structural diversity. In step ②, SCor instantiates a selected skeleton S into concrete queries. Low-cost placeholders are filled with inexpensive values (e.g., constants or small/empty tables), whereas high-cost placeholders are assigned expensive values (e.g., large tables or derived expressions), producing an unbalanced short-circuit query Q_1.

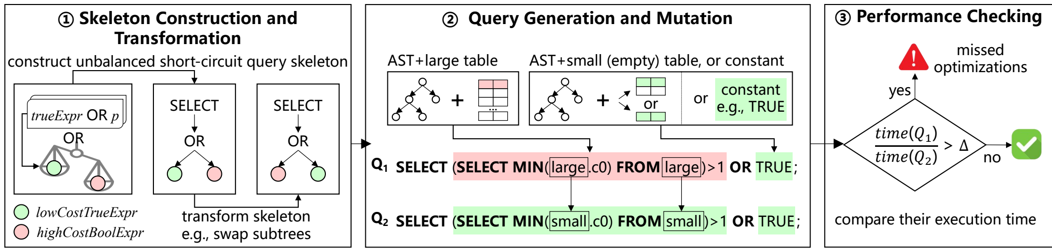

Fig. 2. Overview of SCor workflow. The process consists of three steps: (1) constructing unbalanced short-circuit query skeletons, (2) instantiating skeletons into concrete queries with low- and high-cost placeholders, and (3) executing the queries to detect potential missed optimizations by comparing runtime differences.

During the execution of Q_1, high-cost expressions are expected to be parsed but not fully evaluated. The oracle query Q_2 is then generated by replacing all large table references with small or empty tables of the same schema. In step ③, SCor executes Q_1 and Q_2 and compares their execution times. Because Q_1 is designed to short-circuit, its runtime should closely match that of Q_2, and any substantial deviation signals a potential missed optimization in the DBMS.

3.1 Representative Short-Circuit Patterns

SCor systematically uncovers missed optimization opportunities by constructing unbalanced queries that leverage short-circuit evaluation. To provide the foundation for this approach, we identify five categories comprising ten representative short-circuit query patterns commonly found in official documentation [8, 26, 29, 35, 36, 40] of popular DBMSs (e.g., Oracle, MySQL, MariaDB). These patterns represent optimizations that are either already implemented or are under active consideration by DBMS vendors. The relevance of these patterns is evidenced by direct feedback from developers; for instance, regarding the limited-scan pattern [9], CockroachDB stated: "*The team agrees that this missing optimization would be valuable. We'll prioritize it soon.*"

AND/OR Patterns. The AND and OR logical operators are ubiquitous in SQL expressions, and DBMSs commonly exploit short-circuit evaluation to optimize their execution [36]. For a true expression $trueExpr$, a false expression $falseExpr$, and an arbitrary Boolean expression p, we summarize two short-circuit patterns:

Pattern 1.1 $trueExpr$ OR p
Pattern 1.2 $falseExpr$ AND p

As shown in Equations 1-2, the expression $trueExpr$ OR p is logically equivalent to TRUE, and $falseExpr$ AND p is logically equivalent to FALSE. Consequently, the evaluation of p can be skipped. Such AND/OR expressions frequently appear in many SQL clauses (e.g., ON, WHERE, HAVING, SELECT, functions). Their short-circuit evaluation can significantly impact the overall query performance.

$$trueExpr \text{ OR } p = \text{TRUE}, \quad p \text{ OR } trueExpr = \text{TRUE} \tag{1}$$

$$falseExpr \text{ AND } p = \text{FALSE}, \quad p \text{ AND } falseExpr = \text{FALSE} \tag{2}$$

Conditional Branch Patterns. Conditional branches, such as the CASE expression [35] and the IF() function [48], are widely used in DBMSs. Many systems, such as Oracle and SQLite, have implemented short-circuit evaluation for these expressions [35, 48]. These patterns capture queries where a Boolean condition determines which branch of an expression to evaluate, allowing the system to skip the evaluation of the unneeded branch once the condition's outcome is determined.

Pattern 2.1 IF(p, $expr_1$, $expr_2$)
Pattern 2.2 CASE WHEN p THEN $expr_1$ ELSE $expr_2$ END

Patterns 2.1 and 2.2 are semantically equivalent. Let p be an arbitrary Boolean expression. When p evaluates to TRUE, the evaluation of $expr_2$ can be skipped; conversely, when p evaluates to FALSE, the evaluation of $expr_1$ can be skipped.

$$\begin{aligned} \texttt{IF}(p,\ expr_1,\ expr_2) &= \texttt{CASE WHEN}\ p\ \texttt{THEN}\ expr_1\ \texttt{ELSE}\ expr_2\ \texttt{END} \\ &= \begin{cases} expr_1 & \text{if } p \text{ is } \texttt{TRUE} \\ expr_2 & \text{if } p \text{ is } \texttt{FALSE} \text{ or } \texttt{NULL} \end{cases} \end{aligned} \tag{3}$$

Skipped/Limited Scan Patterns. This category describes optimizations where a DBMS can avoid an expensive full table scan by recognizing that the query logically requires only a limited number of rows, or even none at all. This is a common strategy for efficiently processing top-k selection queries [3] or CREATE TABLE AS SELECT (CTAS) statements [40]. We generalize these scenarios under the name Skipped/Limited Scan Patterns.

Pattern 3.1 LIMIT 0, ON|WHERE|HAVING FALSE
Pattern 3.2 LIMIT n

Pattern 3.1 frequently occurs in CTAS statements [40], which create a new table based on the output of a SELECT query. For instance, to create a table that inherits the schema of existing tables t0 and t1 without copying any data, one can execute SELECT * FROM t0, t1 LIMIT 0. In this case, the DBMS can short-circuit the execution and create the table without scanning any data on disk. Pattern 3.2 commonly occurs when users sample a small number of rows (e.g., 1) from an existing table. For example, SELECT * FROM t1 LIMIT 1 allows the DBMS to perform a limited scan, reading only the necessary rows to satisfy the query rather than scanning the entire table.
FirstMatch Patterns. The FirstMatch strategy, which returns a result upon finding the first valid match rather than processing all possibilities, is widely adopted in DBMSs to improve query execution efficiency. For example, it underlies operations such as the COALESCE() function [8] and SEMI JOIN [26], both leveraging this principle to avoid unnecessary computations

Pattern 4.1 COALESCE()
Pattern 4.2 SEMI JOIN

The COALESCE() function returns the first non-NULL value from a list of arguments, or NULL if all arguments are NULL. It employs short-circuit evaluation: once the first non-NULL value is found, the remaining arguments are not evaluated. Similarly, FirstMatch serves as an execution strategy for semi-join subqueries, where the system executes the subquery and terminates execution immediately upon finding the first matching record. To test whether the semi-join operator short-circuits after the first match—independent of data layout—we insert identical values at different scales into small and large tables. For example, we insert a single row with the value "1" in the small table, and one billion rows with the value "1" in the large table. This setup guarantees an immediate match from the start, independent of data properties (e.g., row order). Therefore, if short-circuit evaluation is implemented, the total execution time of semi-join should be similar to scanning the small table, thereby testing the behavior.
Empty Input Patterns. Empty inputs may arise during query execution [2], either from empty base tables or from intermediate result sets. In such cases, the output is deterministically empty, allowing the optimizer to safely bypass the evaluation of subsequent operators. This category

captures scenarios where the input to a SQL operation is empty, providing opportunities for the DBMS to short-circuit execution and skip unnecessary computations.

Pattern 5.1 *emptyTable* `JOIN` *anyTable*

Short-circuit evaluation occurs when a `SELECT` query references an empty table. In SQL, tables in the `FROM` clause are processed first, followed by `JOIN`, `WHERE`, `GROUP BY`, `HAVING`, `SELECT`, `ORDER BY`, and `LIMIT` clauses. If any table in the `FROM` clause is empty, many subsequent operations can be safely skipped during execution. For example, join operations (ψ) such as `CROSS JOIN`, `INNER JOIN`, `NATURAL JOIN`, `SEMI JOIN` produce an empty result when one input table is empty:

$$\varnothing \,\psi\, T = \varnothing, \quad T \,\psi\, \varnothing = \varnothing \tag{4}$$

`LEFT JOIN` (⟕) and `RIGHT JOIN` (⟖) also yield empty results if the corresponding table is empty:

$$\varnothing \;⟕\; T = \varnothing, \quad T \;⟖\; \varnothing = \varnothing \tag{5}$$

For `ANTI JOIN` (▷), an empty source table produces an empty result:

$$\varnothing \triangleright T = \varnothing \tag{6}$$

Other operations such as `WHERE`, `HAVING`, `GROUP BY`, `ORDER BY`, and `LIMIT` also do not generate rows from an empty input, enabling the DBMS to skip their evaluation.

Pattern 5.2 *emptySet* `INTERSECT|EXCEPT` *anySet*

Query result can be viewed as mathematical sets, allowing set operations like `INTERSECT` and `EXCEPT` to exploit empty inputs for optimization. For any set S, the following identities hold:

$$\varnothing \cap S = \varnothing, \quad S \cap \varnothing = \varnothing \tag{7}$$

$$\varnothing \setminus S = \varnothing \tag{8}$$

Notably, some DBMSs deliberately avoid aggressive compile-time short-circuit due to plan cache [27] considerations. In OLTP systems, execution plans are typically required to be data-independent to enable safe reuse [14]. For Patterns 5.1 and 5.2, eliminating the expensive operand at compile time would require assuming that the emptiness property remains stable; however, this would necessitate frequent plan invalidations when data is updated, thereby undermining the benefits of the plan cache. Consequently, some systems intentionally retain the join or set operation and defer emptiness checks to runtime, trading potential performance gains for plan robustness. We therefore regard these cases as deliberate design choices rather than bugs. To avoid misclassifying such intentional behaviors as performance bugs, we generate queries whose emptiness can be inferred at compile time. Specifically, for Patterns 5.1 and 5.2, we construct provably empty inputs using queries such as `SELECT * FROM table WHERE FALSE` or `SELECT * FROM table LIMIT 0`, all of which allow emptiness to be determined during compilation.

All of the above patterns can be classified into three categories of short-circuit defined in Section 2. Specifically, Pattern 3.1 corresponds to compile-time short-circuit. Constructs such as `LIMIT 0` or predicates like `ON FALSE`, `WHERE FALSE`, and `HAVING FALSE` enable the query optimizer to determine at compile time that no rows will be produced or processed. Consequently, the optimizer can safely eliminate the entire operation during query compilation, before execution begins. Patterns 3.2 (`LIMIT n`) and 4.2 (`SEMI JOIN`) correspond to runtime short-circuit, where execution is terminated early once sufficient results have been obtained or a matching record is found during query execution. Other patterns—including Patterns 1.1, 1.2, 2.1, 2.2, 4.1, 5.1, and 5.2—may exhibit either compile-time or statistics-aware short-circuit, depending on the specific expressions involved and the resulting query structure. For example, the first query in Figure 1 demonstrates a compile-time short-circuit.

However, when `TRUE` is replaced with `(SELECT COUNT(*) FROM small)>1` (assuming table `small` has several rows and the predicate evaluates to `TRUE`) in the same query, the short-circuiting behavior shifts to statistics-aware, since the expression can no longer be resolved at query compilation time and instead depends on metadata, cardinality estimates or specific design choices like plan cache.

3.2 Skeleton Construction and Transformation

To systematically generate queries corresponding to each short-circuit pattern identified in Section 3.1, we design a set of parameterized skeletons. These skeletons are "unbalanced" as they intentionally define placeholders as either low-cost or high-cost items. This setup creates scenarios where evaluating the low-cost item first can determine the final result and, in principle, allows the optimizer to skip the subsequent evaluation of a computationally expensive high-cost item. To further expand the coverage of potential query scenarios, we apply three generic transformation strategies to the abstract grammar tree (AST) of these base skeletons.

Query Skeleton. A query skeleton is a structural template abstracting away database-specific details, thereby focusing on the logical operator composition inherent within SQL queries. It preserves all operational keywords while substituting placeholders for specific database elements like tables, columns, and constant values. These skeletons serve as blueprints for generating concrete test queries capable of exposing missed optimization opportunities.

As illustrated in Table 1, each skeleton contains variables prefixed with $ as placeholders, which are later instantiated with specific tables, columns, expressions, or statements. The placeholders are categorized by their implied computational cost into low-cost or high-cost items. **Low-cost items** include $\$emptyTable$ (an empty table), $\$smallTable$ (a small table, e.g., containing one row), and expressions or statements derived from them, such as $\$lowCostTrueExpr$, $\$lowCostFalseExpr$, and $\$lowCostEmptyStmt$ (e.g., selecting a constant or computing an aggregate over the $\$emptyTable$). **High-cost items** include $\$largeTable$ (a large table, e.g., contains one billion rows) and expressions or statements derived from it, such as $\$highCostBoolExpr$, $\$highCostExpr$, and $\$highCostStmt$ (e.g., computing aggregates over the $\$largeTable$). In addition, some placeholders are constrained to specific values—such as empty set, true, false, NULL, or non-NULL value—to exercise different kinds of short-circuit evaluation scenarios.

Skeleton Transformation. To broaden the coverage of query scenarios, we apply three transformation strategies to the skeletons in Table 1, generating richer variants. As illustrated in Figure 3, these strategies include: swapping low-cost and high-cost items, adding a high-cost item, and a combination of both. Crucially, these transformations are semantics-preserving; they do not change the query's final result, ensuring that short-circuit opportunities remain intact. The validity of the swapping transformation is grounded in the commutative property of operators `OR`, `AND`, `JOIN`, and `INTERSECT`, as formalized in Equations 1, 2, 4, and 7, which allow the positions of operands to be exchanged without affecting the query's semantics. Similarly, the adding transformation is valid because a newly added high-cost item can be logically grouped with an existing one, forming a larger high-cost expression or table whose evaluation is still optimized away by the original short-circuit mechanism. For instance, in the pattern $\$emptyTable$ `JOIN` $\$largeTable$ `JOIN` $\$largeTable$, the two joins with $\$largeTable$ can be regarded as a single, large intermediate table; since the short-circuit is triggered by the leading $\$emptyTable$, the addition of another $\$largeTable$ does not alter the resulting empty set. The combined transformation, integrating both swap and add operations, remains valid for the same underlying reasons. Moreover, these transformations are generic because they operate on the AST, and any query can be transformed into an AST. When adding a new pattern, as long as it conforms to the AST structures used in Figure 3, our transformations can apply directly.

Table 1. Unbalanced short-circuit query skeleton examples of each pattern.

Patterns	Skeleton Examples
P1.1	SELECT $lowCostTrueExpr OR $highCostBoolExpr ;
P1.2	SELECT $lowCostFalseExpr AND $highCostBoolExpr ;
P2.1	SELECT IF($lowCostTrueExpr , $lowCostExpr , $highCostExpr);
P2.2	SELECT CASE WHEN $lowCostTrueExpr THEN $lowCostExpr ELSE $highCostExpr END;
P3.1	SELECT $largeTables.columns FROM $largeTables LIMIT 0 ;
P3.2	SELECT $largeTables.columns FROM $largeTables LIMIT $smallNumber ;
P4.1	SELECT COALESCE($lowCostNotNullExpr , $highCostExpr);
P4.2	SELECT * FROM $smallTable SEMI JOIN $largeTable ON $smallTable.column = $largeTable.column ;
P5.1	SELECT * FROM $emptyTable JOIN $largeTable ;
P5.2	$lowCostEmptyStmt INTERSECT $highCostStmt ;

Variables starting with $ represent placeholders:
$emptyTable, *$smallTable*, and *$largeTable* represent tables with the same schema but different sizes;
$lowCost∗* variables represent low-cost expressions/statements (e.g., constants, calculating aggregate values from *$smallTable);
$highCost∗* variables represent high-cost expressions/statements (e.g., calculating aggregate values from *$largeTable).
Some variables need to satisfy certain constraints, such as "is true", "is false", "is not null", and "is empty".

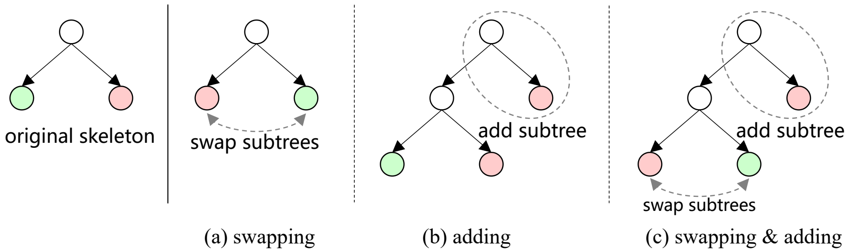

Fig. 3. Three transformation strategies used by SCor for modifying unbalanced short-circuit query skeletons in Table 1, including: (a) swapping, (b) adding, and (c) swapping & adding. ○ represents operators such as OR, AND, JOIN, INTERSECT; ○ represents low-cost items, e.g., *$lowCostTrueExpr*, *$lowCostEmptyStmt*; ○ represents high-cost items, e.g., *$highCostBoolExpr*, *$highCostStmt*.

3.3 Query Generation and Mutation

To construct unbalanced short-circuit queries and their corresponding oracle queries, we proceed in two main steps. First, we populate a skeleton S with concrete values from a given database D to obtain an executable query Q_1. Then, we apply a replacing mutation to the unbalanced short-circuit query Q_1, generating its corresponding oracle query Q_2.

Skeleton Population. As detailed in Table 1, each query skeleton incorporates two distinct item categories: low-cost items and high-cost items. We employ several strategies to ensure high-cost items consistently incur greater execution overhead. First, when executing identical query structures on tables with the same schema but different sizes, operations on larger tables naturally demonstrate higher costs than those on smaller tables. Second, we leverage the fundamental cost difference between constants and variables. For instance, the predicate `t0.c0 = $expr` (e.g., *$expr* is a scalar subquery returning an undermined value) is more expensive to compute than `t0.c0 = 1`. Beyond these core approaches, other strategies can further differentiate costs, such as incorporating additional complex operations. For example, `SELECT * FROM t1 ORDER BY t0.c0` incurs a higher execution cost than the simpler `SELECT * FROM t1`, as the former performs an additional sort

operation. To systematically materialize these cost-aware skeletons into executable queries, we employ the population algorithm detailed below.

Algorithm 1: Skeleton population

Input : S: an unbalanced short-circuit query skeleton
D: a database has three tables of identical schema:
an empty table $emptyTable$
a small table $smallTable$ (e.g., has one row)
a large table $largeTable$ (e.g., billions rows)
Output: Q_1: an unbalanced short-circuit query

```
1  Procedure PopulateSkeleton(S, D):
2      items ← GetToPopulateItems(S);
3      foreach item in items do
4          if item ∈ lowCostItems then
5              item.value ← GenerateValue(item.constraint, emptyTable, smallTable,
                 constantSet);
6              if item.value is Boolean then
7                  item.value ← AlgebraicEquivalenceTransformation(item.value)
8          else
9              item.value ← GenerateValue(item.constraint, largeTable);
10     return Q1 ;
```

As outlined in Algorithm 1, SCor takes as input a short-circuit query skeleton S and a database D containing tables of identical schema but varying sizes, then outputs an unbalanced short-circuit query Q_1. The algorithm begins by identifying all placeholder elements within the skeleton that require concrete values (line 2). During population, SCor strategically assigns concrete values based on execution cost requirements. Specifically, items designated as low-cost are populated with values from either the empty table or small table, or assigned constant values (e.g., `TRUE`, `FALSE`), to ensure minimal execution time (lines 4-5). In addition, we apply algebraic equivalence transformations [5] to the generated Boolean value (lines 6-7) to assess query optimization behavior further. Concretely, we replace literal constants with provably equivalent non-literal expressions, such as transforming `TRUE` into `2 > 1` or `(SELECT COUNT(*) FROM t0)>= 0`, and `FALSE` into `1 = 2` or `a > 3 AND a < 2`. This allows us to evaluate whether the optimizer can correctly reason about equivalence through constraint propagation, compile-time expression simplification, and related rewrite and inference mechanisms. Conversely, the remaining items are assigned values from the large table to form expensive execution branches (lines 8-9). All value assignments strictly adhere to each item's constraint specifications. For instance, when an item requires an empty result set (e.g., P5.1 in Table 1), values are sourced from the empty table to guarantee zero matches. The final populated skeleton yields an unbalanced short-circuit query Q_1 (line 10), specifically designed to expose missed optimization opportunities in DBMSs.

Oracle Query Generation. For an unbalanced short-circuit query Q_1, we expect that the DBMS only parses but does not evaluate the high-cost items during execution. To generate an oracle query Q_2 that validates this expectation, we replace all table names in high-cost expressions and statements (originally referencing the large table) with corresponding empty or small table names of the same schema, as shown in Figure 2. This transformation minimizes the execution cost of the

high-cost components—preserving only the parsing overhead—while simulating the ideal scenario where high-cost evaluation is skipped. We expect Q_1 to demonstrate similar performance to Q_2 when optimizations are properly applied.

3.4 Performance Checking

To identify missed optimization opportunities, we compare the execution time of the unbalanced short-circuit query Q_1 with that of its corresponding oracle query Q_2. As shown in Equation 9, when optimizations are correctly applied, the execution time of Q_1 should be close to that of Q_2, yielding a bounded performance ratio:

$$time(Q_1) \approx time(Q_2) \rightarrow \frac{time(Q_1)}{time(Q_2)} < \Delta \tag{9}$$

Algorithm 2: Performance checking

```
Input  : Q1: unbalanced short-circuit query
         Q2: corresponding oracle query of Q1
         Δ: the threshold of time(Q1)/time(Q2)
         N: repeat times to confirm the difference
Output: missed optimization reports
1  Procedure CheckPerformanceDiff(Q1, Q2):
2      time(Q2) ← ExecuteQuery(Q2);
3      time(Q1) ← ExecuteQueryWithTimeout(Q1, ⌈time(Q2) × Δ⌉);
4      if time(Q1) / time(Q2) >= Δ then
5          if Confirm(Q1, Q2, N) then
6              GenReport(Q1, Q2, time(Q1), time(Q2)) ;
7  Procedure ExecuteQueryWithTimeout(Q, T):
8      if time(Q) > T then
9          TerminateQuery(Q);
10         time(Q) ← T ;
11     return time(Q);
12 Procedure Confirm(Q1, Q2, N):
13     count ← 1 ;
14     for k ← 1 to N − 1 do
15         time(Q2) ← ExecuteQuery(Q2);
16         time(Q1) ← ExecuteQueryWithTimeout(Q1, ⌈time(Q2) × Δ⌉);
17         if time(Q1) / time(Q2) >= Δ then
18             count ← count + 1;
19     if count = N then
20         return True ;
```

As outlined in Algorithm 2, SCor takes an unbalanced short-circuit query Q_1 and its corresponding oracle query Q_2 as input and produces performance bug reports when significant slowdowns are detected. The core of SCor 's process is the `CheckPerformanceDiff` function (lines 1–6), which

compares the execution times of Q_1 and Q_2. Specifically, SCor first executes the oracle query Q_2 and records its execution time, denoted as $time(Q_2)$ (line 2). Since some randomly generated unbalanced queries (Q_1) may execute inefficiently due to missed optimizations, a timeout mechanism is applied (lines 7–11): any query exceeding a predefined execution threshold ($\mathcal{T}$) is terminated, and its execution time is set to $\mathcal{T}$. We set $\mathcal{T} = \lceil time(Q_2) \times \Delta \rceil$ (line 3), where Δ is a configurable slowdown factor. This adaptive threshold design eliminates the need for DBMS-specific tuning of absolute time limits, as the threshold adapts dynamically based on the value of $time(Q_2)$. If Q_1 is significantly slower than Q_2 by more than a ratio threshold Δ (line 4), it is flagged as a potential missed optimization. To mitigate false positives from system noise such as cache effects [27], SCor invokes the `Confirm` function (line 5), which executes both queries N times in isolation and in different execution orders, and checks whether the slowdown consistently persists (lines 12-20). Once confirmed, SCor generates a bug report (line 6), containing the query pair and their corresponding execution times.

The substantial performance difference between Q_1 and Q_2 provides clear guidance for threshold configuration. Since Q_2 executes only low-cost items, its runtime is typically minimal, often completing in milliseconds. In contrast, when a missed optimization exists in Q_1, it executes expensive items that require seconds, minutes, or longer to complete. This pronounced performance gap, evident in cases like the 2995× degradation shown in Figure 1, informs our threshold selection methodology. Accordingly, we set Δ to a high value based on empirical experience and developer feedback. This configuration ensures reliable detection of optimization misses.

4 Implementation

We implemented SCor as a DBMS testing tool comprising approximately 5k lines of Python code. In this section, we present the technical details for implementing SCor.

Database State Generation. To construct unbalanced short-circuit queries, SCor creates tables with varying data sizes, including several empty tables ($emptyTable), small tables ($smallTable) containing a limited number of records (e.g., 1-100 rows), and large tables ($largeTable) with a substantial volume of data (e.g., millions or billions of rows). This enables the generation of both high-cost expressions ($highCostExpr), such as queries executed on large tables, and low-cost expressions ($lowCostExpr), such as those applied to small or empty tables.

Given the wide array of existing data generation methods—such as testing benchmarks, TPC-H [52], automated testing tools like SQLancer [42–44], and built-in DBMS functions [41]—SCor does not reimplement data generation. Instead, it leverages these established approaches. For instance, in PostgreSQL, the `generate_series()` function can be invoked to produce sequences of varying sizes, facilitating the creation of tables with diverse data volumes.

Query Generation. SCor instantiates the query skeletons in Table 1 by generating concrete SQL expressions and statements based on the initial database state. To ensure syntactic correctness, we employ a grammar-based approach that encodes SQL-99 [13] grammar into production rules. Through random walks over the resulting grammar tree, the system generates diverse valid SQL queries. Table 2 illustrates a grammar subset that enables the generation of various `SELECT` statements and expressions for filling the skeletons. Moreover, the generated statements and expressions must satisfy specific constraints defined in Table 1. For example, low-cost `SELECT` statements or expressions are generated using empty tables, small tables, or constants; high-cost statements or expressions are generated using large tables; and Boolean expressions use constants (e.g., `2<1` for false) or deterministic operations (e.g., `(SELECT COUNT(*) FROM t1)>=1` for true where `t1` is a small table with known row counts) to ensure predictable evaluation outcomes.

Table 2. A subset of the SQL grammar that allows SCor to generate various SELECT statements and expressions.

<table>
<tr><td>$selectStmt</td><td>::=</td><td>SELECT $selectList FROM $table [$joinType JOIN $table [ON $condition]]
[WHERE $condition] [GROUP BY $columnList [HAVING $condition]]
[ORDER BY $columnList [ASC|DESC]] [LIMIT $number]</td></tr>
<tr><td>$selectExpr</td><td>::=</td><td>$operand $comparison ($selectStmt) | ($selectStmt) // scalar query</td></tr>
<tr><td>$selectList</td><td>::=</td><td>$columnList|$aggFunc($column)|*</td></tr>
<tr><td>$columnList</td><td>::=</td><td>$column [, $column]*</td></tr>
<tr><td>$aggFunc</td><td>::=</td><td>COUNT|SUM|AVG|MAX|MIN</td></tr>
<tr><td>$joinType</td><td>::=</td><td>CROSS|FULL|LEFT|RIGHT|INNER|NATURAL|SEMI|ANTI</td></tr>
<tr><td>$condition</td><td>::=</td><td>$operand $comparison $operand [AND|OR condition]*</td></tr>
<tr><td>$operand</td><td>::=</td><td>$column|$constant|($selectExpr)</td></tr>
<tr><td>$comparison</td><td>::=</td><td>>|>=|<|<=|=|!=|LIKE|NOT LIKE|IN|NOT IN|EXISTS|NOT EXISTS|...</td></tr>
</table>

$selectStmt → *$lowCostEmptyStmt*, *$highCostStmt*.
$selectExpr → *$lowCostBoolExpr*, *$lowCostTrueExpr*, *$lowCostFalseExpr*, *$lowCostNotNullExpr*, *$lowCostlExpr*, *$highCostExpr*.

5 Evaluation

To evaluate the effectiveness and efficiency of SCor in detecting missed optimization opportunities in real-world DBMSs, we design the following three questions:

- **Q1. Detection Capability**: How many missed optimization opportunities in real-world DBMSs can SCor detect?
- **Q2. Pattern Analysis**: How effective are these short-circuit patterns used by SCor in detecting missed optimization opportunities in DBMSs?
- **Q3. Comparative Evaluation**: Can SCor achieve superior performance compared to existing DBMS benchmarking methodologies?

5.1 Testing Setup

Tested DBMSs. We applied SCor to 11 widely-used DBMSs and tested the latest release versions of each DBMS, including Oracle 23ai, PostgreSQL 17.6, MySQL 9.4.0, MariaDB 11.8.1, ClickHouse 25.4.2, CockroachDB 24.3.13, TiDB 8.5.1, OpenGauss 7.0.0, OceanBase 5.7.25, SQLite 3.50.1, and DuckDB 1.3.1. Since the latest releases are generally considered advanced, any missed optimization opportunities identified in these DBMSs serve as strong evidence of the effectiveness of our approach. According to the DB-Engine's Ranking [10], these selected DBMSs span diverse categories and represent some of the most popular and widely-used systems. Oracle [34] is a leading commercial relational DBMS, widely regarded for its advanced enterprise features. PostgreSQL [39] is a powerful object-relational DBMS, known for standards compliance and extensibility. MySQL [28] and MariaDB [25] are traditional relational DBMSs. TiDB [50] is a distributed relational DBMS that provides horizontal scalability and high availability for large-scale deployments. CockroachDB [7] is a cloud-native, distributed SQL DBMS designed for resilience and global scalability. ClickHouse [6] is a column-oriented DBMS optimized for real-time analytical processing (OLAP). OpenGauss [33] is an enterprise-level relational DBMS with strong security and reliability features. OceanBase [31] is a distributed relational DBMS developed for high performance in financial and enterprise applications. SQLite [47] and DuckDB [12] are lightweight embedded DBMSs designed for efficient in-process execution without requiring a standalone server.

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

Table 3. SCor found 153 unique missed optimization opportunities across 11 mature DBMSs, of which 125 have been confirmed (33 already fixed), 28 are investigating. "F" indicates the missed optimization can be traced back to the first release, and "FSFS" indicates the missed optimization can be traced back to the first syntax-feature-supported release.

DBMS	First release	Bugs	Short-circuit patterns	Status			Since		Longest latency
				Confirmed	Fixed	Open	F	FSFS	
Oracle	1979	2	**1.1**(1), **1.2**(1)	2	0	0	0	2	1.4 years
PostgreSQL	1996	3	**1.1**(2), **1.2**(1)	3	0	0	0	3	22.3 years
MySQL	1995	13	**1.1**(3), **1.2**(2), **3.1**(2), **5.1**(1), **5.2**(5)	13	0	0	2	9	25.1 years
MariaDB	2009	21	**1.1**(2), **1.2**(2), **3.1**(2), **4.1**(1), **5.2**(14)	13	13	8	6	15	16 years
ClickHouse	2016	12	**1.1**(3), **1.2**(2), **4.1**(1), **5.1**(4), **5.2**(2)	7	2	5	9	3	9.3 years
CockroachDB	2017	6	**3.2**(1), **4.2**(1), **5.1**(2), **5.2**(2)	6	0	0	6	0	8.4 years
TiDB	2017	26	**1.1**(3), **1.2**(2), **2.1**(3), **2.2**(1), **3.1**(3), **3.2**(3), **4.1**(1), **5.1**(1), **5.2**(11)	26	0	0	15	9	8 years
OpenGauss	2020	9	**1.1**(1), **3.1**(3), **5.1**(3), **5.2**(2)	9	5	0	8	0	5.2 years
OceanBase	2021	25	**1.1**(3), **1.2**(2), **2.1**(1), **2.2**(1), **4.1**(1), **5.1**(14), **5.2**(3)	12	0	13	25	0	4 years
SQLite	2000	21	**1.1**(3), **1.2**(4), **5.1**(11), **5.2**(3)	19	13	2	2	18	25 years
DuckDB	2019	15	**1.1**(4), **1.2**(4), **2.1**(1), **2.2**(1), **3.1**(2), **4.1**(1), **5.2**(2)	15	0	0	14	1	6.3 years
Total	-	153	**1.x**(45), **2.x**(8), **3.x**(14), **4.x**(6), **5.x**(80)	125	33	28	87	60	25.1 years

5.2 Detection Capability

Overall Result. As shown in Table 3, SCor has detected a total of 153 previously unknown missed optimization opportunities on 11 well-tested DBMSs. Of them, 125 bugs were confirmed, 33 have been fixed, and 28 are being investigated, including 2 in Oracle, 3 in PostgreSQL, 13 in MySQL, 21 in MariaDB, 12 in ClickHouse, 6 in CockroachDB, 26 in TiDB, 9 in OpenGauss, 25 in OceanBase, 21 in SQLite, and 15 in DuckDB. Pattern analysis reveals that P5.x and P1.x are the most prevalent, accounting for 80 and 45 instances, respectively. This dominance is due to their ability to interact with a wide range of SQL features. For instance, the empty input after `FROM` (pattern 5.1) can affect subsequent clauses such as `JOIN`, `WHERE`, `GROUP BY`, `HAVING`, `SELECT`, `ORDER BY`, and `LIMIT`. Similarly, `OR`/`AND` expressions (patterns 1.1 and 1.2) can appear in various SQL clauses such as `ON`, `WHERE`, `HAVING`, `SELECT`, making them particularly pervasive in real-world queries.

Confirming and Fixing Statistics. Of the 153 reported issues, 81.70% were confirmed, 21.57% have already been fixed. Compared with prior work such as AMOEBA [24], which reports confirmation and fix rates of 15.38% and 12.82%, respectively, our substantially higher rates underscore both the severity of the identified issues and the strong level of attention they have received from DBMS developers. Confirming and fixing these issues is non-trivial. Many bugs are inherently complex, involving intricate interactions among multiple system components, which significantly complicates root-cause analysis. As a result, developers require more time for analysis, testing, and remediation. The most complex fix [49] addressed a bug in SQLite, which introduced optimizations to detect early when queries return no rows due to empty tables. Specifically, the fix implemented an `EXISTS`-to-`JOIN` transformation that converts `EXISTS` constraints into additional terms in the `FROM` clause. Its complexity stems from the interaction of multiple modules—including the optimizer, and executor—and needs to correctly handle different join types (e.g., `OUTER JOIN`, `INNER JOIN`, `CROSS JOIN`). This change required modifications across 13 different files and 204 lines of code.

Latency of the Missed Optimizations Found by SCor. As shown in Table 3, the missed optimizations discovered by SCor represent deeply rooted issues in DBMS codebases. Of the 153 identified issues, 87 have existed since the initial release of their respective systems, while 60 date back to the first release supporting the relevant SQL syntax (e.g., Oracle 23ai introduced support for `SELECT` expressions without `FROM` clause). Only 6 were regression issues introduced later. The longevity is remarkable: the oldest bug persisted for 25.1 years in MySQL, while PostgreSQL, MySQL,

> "xxx...I'd like to write a blog on your contributions...
> I really do appreciate the bug reports and the thinking that can generate better queries for MariaDB."
>
> MariaDB Chief Innovation Officer

> "The whole exercise is pretty questionable really, considering how weak our cost model for expressions is."
>
> PostgreSQL Developer

> "@xxx. Thanks for the report! The team agrees that this missing optimization would be valuable. We'll prioritize it soon."
>
> CockroachDB Developer

> "@xxx. Thank you for your feedback. This issue will be handled as a requirement and optimized in future versions."
>
> Oceanbase Developer

> "... Bummer. That is now on my list of things to fix. I'm glad it was discovered before the 3.51.0 release."
>
> SQLite Developer

Fig. 4. Some feedback from DBMS developers.

and SQLite all contained undetected optimization misses for over two decades. Even newer systems like CockroachDB, TiDB, and OceanBase harbored these issues for 4-8 years. Moreover, for some established vendors like MySQL, whose initial release dates to 1995, the earliest available version in open-source repositories is from 2000, which somewhat constrains our ability to trace bugs to the earlier release. These findings demonstrate SCor's effectiveness in uncovering non-trivial, long-latent optimization opportunities that have consistently evaded existing detection methods.

Feedback From DBMS Developers. We received positive feedback from developers, which not only validated our efforts but also indicated their commitment to fix these bugs, as illustrated in Figure 4. The MariaDB Chief Innovation Officer noticed our bug reports and wrote a thank-you letter to us: " ...I'd like to write a blog on your contributions...I really do appreciate the bug reports and the thinking that can generate better queries for MariaDB." A PostgreSQL developer acknowledged the missed optimization we found, noting, "considering how weak our cost model for expressions is." The developers from CockroachDB, OceanBase, and SQLite confirmed the value of the identified missed optimization opportunities and indicated plans to fix them.

Listing 1. Illustrating significant performance degradation from a missed optimization in SQLite, as detected by SCor. The fix reduces execution time from over 2 days to 0.001s.

```
CREATE TABLE t0 (c0 INTEGER );
SELECT CAST(SUM(t0.c0) AS REAL), COUNT(*) OVER ( ROWS BETWEEN 0 FOLLOWING AND '
     9223372036854775807' FOLLOWING) FROM t0;
-- Prior to optimization: runs for over 2 days without completion
-- Post-optimization: returns empty result in 0.001s
```

Significant Performance Degradation. The missed optimizations identified by SCor can lead to severe performance degradation. This occurs because the DBMS unnecessarily executes expensive operations—such as scanning large tables or computing complex expressions—that should have been optimized away. Moreover, the severity of this degradation is directly proportional to the cost of the unoptimized elements in unbalanced short-circuit queries, meaning the performance impact has no inherent upper bound. Listing 1 illustrates a concrete case instance of the empty input pattern, where a missed short-circuit optimization in SQLite leads to extreme performance

loss. To detect this bug, we followed three steps. First, we select the empty input pattern and construct an empty table. Second, we generate a query that reads from this empty table and applies a window function with a frame defined as: `COUNT(*) OVER (ROWS BETWEEN 0 FOLLOWING AND '9223372036854775807' FOLLOWING)`. Third, we execute the generated query and observe its runtime behavior. This window function aggregates values from the current row to the last row. Because the input table is empty, the query should terminate immediately and return an empty result. However, due to a bug in SQLite, the engine still attempts to allocate and process the extremely large window frame, even though no rows are present. As a result, the query runs for an excessively long time—continuing for over 2 days without terminating until manually canceled.

Value of the Missed Optimizations Found by SCor. Oracle serves as a compelling case study for assessing the value of the missed optimizations detected by SCor. As one of the most advanced commercial DBMSs with decades of refinement, Oracle incorporates a highly sophisticated optimizer. The fact that SCor uncovered only 2 missed optimizations in Oracle is particularly revealing: the majority of unbalanced short-circuit queries executed efficiently, demonstrating Oracle's successful implementation of these optimizations. This outcome validates that the missed optimizations detected by SCor represent essential performance enhancements already adopted by an industry-leading DBMS [34]. Consequently, their absence in other DBMSs indicates significant and valuable improvement opportunities.

5.3 Pattern Analysis

To demonstrate the effectiveness of the patterns we summarized in detecting missed optimization opportunities in DBMSs, we analyze the bugs and present the following case studies.

`OR` Pattern. Listing 2 shows a missed optimization we found in PostgreSQL. In this case, the predicate `t0.c0 > 0` is significantly cheaper to evaluate than the subquery `t0.c0 IN (SELECT * FROM t1 WHERE t1.c1 = t0.c0)`. The first query executes efficiently in 0.139 seconds because PostgreSQL's short-circuit evaluation correctly evaluates the left operand (`t0.c0 > 0`, which is true) first and skips the expensive subquery. However, in the second query, although short-circuit evaluation is available, the optimizer fails to perform cost-based expression reordering to position the cheaper `c0 > 0` predicate first. Instead, as evidenced by the execution plan showing "`Filter: ((ANY (c0 = (SubPlan 1).col1)) OR (c0 > 0))`", it prioritizes the expensive subquery. This forces the system to begin with the costly subquery execution, requiring a full table scan and resulting in a significantly longer execution time of 6.222 seconds.

Listing 2. A missed optimization opportunity of `OR` pattern identified by SCor in PostgreSQL.

```
CREATE TABLE t0(c0 INT8); -- Insert t0 with values: 1-10. Therefore, t0.c0 > 0 is true
CREATE TABLE t1(c1 INT8); -- Insert t1 with values: 1-1000000

SELECT * FROM t0 WHERE t0.c0 IN (SELECT * FROM t2 WHERE t2.c2 = t0.c0) OR t0.c0 > 0; -- 6.222s
 Seq Scan on t0  (...) -- This plan is less efficient
   Filter: ((ANY (c0 = (SubPlan 1).col1)) OR (c0 > 0))...

SELECT * FROM t0 WHERE t0.c0 > 0 OR t0.c0 IN (SELECT * FROM t1 WHERE t1.c1 = t0.c0); -- 0.139s
 Seq Scan on t0  (...) -- This plan is more efficient
   Filter: ((c0 > 0) OR (ANY (c0 = (SubPlan 1).col1)))...
```

Listing 3 shows another `OR`-pattern case where the optimizer pulls up a correlated subquery and reorders the evaluation accordingly. This rewrite biases cost-based exploration toward a join-based plan and causes the optimizer to prune the alternative plan that evaluates the selective predicate `t0.c0 > 0` first. As a result, the optimizer rejects a more efficient plan that enables short-circuit evaluation and instead executes an expensive subquery eagerly, leading to a 918× performance

degradation. The issue was ultimately traced to a default configuration option, `rewrite_rule = 'enable_pullup_expr_sublink'`, which causes OpenGauss to evaluate subqueries eagerly. Given that OpenGauss provides various of configuration options, isolating this particular setting and tracing its impact was especially challenging.

Listing 3. A case of rejected plan reordering in OpenGauss which leads to a 918× performance degradation.

```
CREATE TABLE t0(c0 INT8); -- Insert t0 with values: 1-100
CREATE TABLE t1(c1 INT8); -- Insert t1 with values: 1-1000000
SELECT t0.c0 FROM t0 WHERE t0.c0 > 0 OR EXISTS (SELECT 1 FROM t1 WHERE t1.c1 = t0.c0);

-- More expensive plan: pull up subquery and evaluate it first; consume 1256.294ms
Hash Right Join  (...) (...)
  Hash Cond: (t1.c1 = t0.c0)
  Filter: ((t0.c0 > 0) OR (t1.c1 IS NOT NULL))
  -> HashAggregate (...)
         Group By Key: t1.c1
         ->  Seq Scan on t1  (...) (...)
  -> Hash (...) (...)
         ->  Seq Scan on t0  (...)

-- More efficient plan: Seq Scan on t0 and evaluate t0.c0 > 0 first; consume 1.369ms
Seq Scan on t0  (...)
  Filter: ((c0 > 0) OR (alternatives: SubPlan1 or hashed SubPlan2))
  SubPlan1
     -> Seq Scan on t1 (...) (...never executed)
         Filter: (c1 = t0.c0)
  SubPlan2
     -> Seq Scan on t1 (...) (...never executed)
```

AND Pattern. Listing 4 illustrates a performance regression bug in MySQL 9.4.0 triggered by a short-circuit query. Here, `t0` is a small table and `t1` is a large table. Since MySQL already implements short-circuit evaluation for `AND` expressions in the `WHERE` clause, evaluating the highly selective predicate `t0.c0 < 1` first allows the second predicate `t0.c0 IN (SELECT * FROM t1 WHERE t1.c0 = t0.c0)` to be short-circuited, thereby avoiding expensive scans on the large table `t1`. The query executes almost instantly (0.01 seconds) in MySQL 5.5.27 but takes 4.866 seconds in MySQL 9.4.0, indicating a clear performance regression in the newer version.

Listing 4. A performance regression bug in MySQL 9.4.0 for a short-circuit query, where execution time degrades by over 486× compared to MySQL 5.5.27

```
CREATE TABLE t0(c0 INT8);-- Insert t0 with values: 1-100
CREATE TABLE t1(c0 INT8);-- Insert t1 with values: 1-10000000

-- MySQL 5.5.27: 0.01s; MySQL 9.4.0: 4.866s
SELECT * FROM t0 WHERE t0.c0 < 1 AND t0.c0 IN (SELECT * FROM t1 WHERE t1.c0 = t0.c0);
```

Conditional Branch Pattern. Listing 5 shows a missed optimization in OceanBase involving conditional evaluation in `CASE` expressions. We construct an unbalanced short-circuit query where the `WHEN` branch contains `TRUE`, the `THEN` branch returns the constant `1`, and the `ELSE` branch contains the expensive subquery `SELECT COUNT(*) FROM t1`, which performs a full scan on the large table `t1` containing 1 million rows. This query should always return `1` immediately since the condition is always true, making the `ELSE` branch evaluation unnecessary. Thus, the subquery should be eliminated at compile time, without being planned or executed. To establish the expected performance baseline, we create an oracle query by replacing `t1` with the empty table `t0` in the `ELSE` branch, which

executes quickly. The missed optimization occurs because OceanBase fails to apply short-circuit evaluation to the `CASE` expression when handling subqueries in the unreachable branch.

Listing 5. A missed optimization opportunity of conditional branch pattern identified by SCor in OceanBase.

```
CREATE TABLE t0(c0 INT); -- Empty table
CREATE TABLE t1(c1 INT); -- Large table with 1 million rows
-- Q1's ELSE branch is expensive due to scanning t1, taking 0.148s
SELECT CASE WHEN TRUE THEN 1 ELSE (SELECT COUNT(*) FROM t1) END;
-- Q2 replaces t1 with t0 in Q1, executing quickly in 0.001s
SELECT CASE WHEN TRUE THEN 1 ELSE (SELECT COUNT(*) FROM t0) END;
```

Limited Scan Pattern. Listing 6 shows a bug we detected in CockroachDB. In this case, the optimizer should apply a limited table scan when a query contains a `LIMIT` clause, since only a small number of rows are required to produce the result. For a single-table query with `LIMIT 1`, the optimizer correctly performs an efficient limited scan. However, when the query involves a `CROSS JOIN` operation between two large tables, the optimizer fails to apply the same optimization. It unnecessarily performs a full Cartesian product of both tables before applying the limit, leading to a severe performance degradation of over 668 seconds for a query that should complete in milliseconds. A more efficient strategy would allow the executor to return a result as soon as it encounters the first pair of rows from `t1` and `t2`. Consequently, scanning a single row from each table is sufficient, eliminating the need to materialize the full Cartesian product. After we reported this issue, CockroachDB developers implemented the optimization, reducing the execution time to 0.02 seconds by applying a limited scan.

Listing 6. A missed optimization opportunity of limited scan pattern identified by SCor in CockroachDB.

```
CREATE TABLE t1(c1 INT8); -- Insert into t1 with 1 billion rows
CREATE TABLE t2(c2 INT8); -- Insert into t2 with 1 billion rows
SELECT * FROM t1 LIMIT 1; -- 0.008s, perform limited scan
-- Prior to optimization, CockroachDB performs a full scan:
SELECT * FROM t1 CROSS JOIN t2 LIMIT 1; -- 668.884s
-- Post-optimization, CockroachDB performs a limited scan:
SELECT * FROM t1 CROSS JOIN t2 LIMIT 1; -- 0.02s
```

Listing 7. A missed optimization opportunity of FirstMatch pattern identified by SCor in TiDB.

```
CREATE TABLE t1(c1 INT8); -- Insert into t1 with 1 billion rows
SELECT COALESCE(null, 1); -- 0.001s
SELECT COALESCE(null, 1, (SELECT MAX(c1) FROM t1)); -- 101.473s
EXPLAIN SELECT COALESCE(null, 1, (SELECT MAX(c1) FROM t1)); -- 99.841s
```

FirstMatch Pattern. Listing 7 reveals a missed optimization opportunity in TiDB concerning the `COALESCE()` function. Since `COALESCE()` returns the first non-NULL argument, many DBMSs adopt a short-circuit evaluation strategy [8] that stops once the first non-NULL argument is found. Here, the second argument "1" provides a valid non-NULL value, making evaluation of the subquery in the third argument unnecessary. Consequently, the query result can be determined at compile time without any runtime execution. However, TiDB fails to apply this short-circuit optimization and instead executes the expensive subquery against the large table, resulting in performance degradation exceeding 100 seconds for a query that should complete in milliseconds. Furthermore, the `EXPLAIN` statement also incurs similar overhead by unnecessarily executing scalar subqueries during query analysis, which is unexpected since `EXPLAIN` statements typically should not execute query components [37]. This behavior indicates that TiDB eagerly evaluates subqueries, even when the query result is already statically determined, leading to unnecessary performance costs.

Empty Input Pattern. Listing 8 illustrates two performance bugs detected by SCor in TiDB. All queries involve operators where the emptiness of one input—introduced by a `WHERE FALSE` predicate—can be determined at compile time. When the empty result set appears on the favorable side of a `JOIN` or as the first operand of `INTERSECT`, TiDB successfully short-circuits query execution, yielding response times close to 0.001 seconds. However, for a `LEFT JOIN`, placing the empty table on the left-hand side, or for an `INTERSECT`, placing it as the second operand, should likewise produce an immediate empty result. Instead, TiDB fails to apply the same compile-time optimization in these syntactically different yet logically equivalent scenarios. As a result, it unnecessarily performs a full scan of table `t1`, causing execution times to increase to several seconds. This asymmetry in optimization behavior highlights a missed opportunity for more comprehensive query simplification, which SCor is able to effectively identify.

Listing 8. Two missed optimization opportunities of empty input pattern identified by SCor in TiDB using queries whose emptiness can be inferred at compile time.

```
CREATE TABLE t1(c1 INT8); -- Insert into t1 with 10 million rows

-- Pattern 5.1: emptyTable JOIN anyTable
SELECT * FROM t1 RIGHT JOIN (SELECT * FROM t1 WHERE FALSE) AS e ON e.c1 = t1.c1; -- 0.001s
SELECT * FROM (SELECT * FROM t1 WHERE FALSE) AS e LEFT JOIN t1 ON e.c1 = t1.c1; -- 3.975s

-- Pattern 5.2: emptySet INTERSECT anySet
SELECT * FROM t1 WHERE FALSE INTERSECT SELECT * FROM t1; -- 0.001s
SELECT * FROM t1 INTERSECT SELECT * FROM t1 WHERE FALSE; -- 18.764s
```

5.4 Comparative Evaluation

To illustrate the effectiveness and efficiency of our tools, we conducted a comparative evaluation against existing DBMS performance bug testing tools.

Baseline. We compared SCor with five state-of-the-art DBMS performance bug detection tools, including: APOLLO [18], AMOEBA [24], and CERT [1], Puppy [54], and HULK [55]. APOLLO detects performance regression bugs by comparing the execution time of the same query across different DBMS versions. AMOEBA detects performance bugs by comparing execution times of semantically equivalent queries. CERT tackles performance bugs through cardinality estimation analysis. Puppy detects the performance bug by changing the optimization configurations. HULK detects the data-sensitive performance anomalies by changing the data volume.

Effectiveness of SCor Comparison to Baseline. In this experiment, we run each tool on the latest version of each supported DBMS for a duration of 24 hours. Since the baseline approaches do not support all the DBMSs that SCor does, our comparison focuses on the DBMSs commonly supported by both SCor and each baseline. Table 4 presents the number of unique performance bugs identified by each approach on each DBMS. The results show that SCor found 24, 8, 36, 8, and 4 more bugs than APOLLO, AMOEBA, CERT, Puppy, and HULK, respectively. Note that the increments are calculated based only on the DBMSs commonly supported by both approaches. The other approaches were less effective in our experiments because we tested the latest versions of the DBMSs, where previously known bug patterns have largely been addressed. This reduced effectiveness suggests that discovering additional bugs has become more challenging for the baseline, and the bugs detected by our approach are likely beyond the reach of existing approaches.

Furthermore, our analysis reveals that the missed optimizations identified by SCor show limited overlap with only 4 bugs (1 PostgreSQL bug and 3 MySQL bugs overlap) in common with HULK and no overlap with other baseline tools. This partial alignment with HULK is expected, as both approaches leverage data volume variation. However, their testing strategies differ: HULK detects

Table 4. Number of unique performance bugs detected by each testing tool in 24 hours. This comparison focuses on the DBMSs commonly supported by both SCor and each baseline. "-" indicates that the tool does not support the DBMS. "Increments" are calculated only for commonly supported DBMSs.

DBMS	APOLLO	AMOEBA	CERT	Puppy	HULK	SCor
PostgreSQL	0	0	-	2	2	3
MySQL	-	-	3	6	10	13
CockroachDB	-	1	2	-	-	6
TiDB	-	-	4	-	-	26
SQLite	0	-	-	-	-	21
Total	0	1	9	8	12	69
Increment	24 ↑	8 ↑	36 ↑	8 ↑	4 ↑	-

data-sensitive performance anomalies through data changes, while SCor employs unbalanced short-circuit queries. The limited overlap with HULK confirms that SCor's pattern-based generation strategy uncovers different optimization opportunities, systematically constructing queries with short-circuit patterns that HULK does not target.

The non-overlap with other tools, except HULK, stems from fundamental differences in both target and methodology. (1) Target: Existing baselines primarily target performance degradation in implemented optimizations, whereas SCor specifically detects missed optimization. Specifically, APOLLO targets performance regression bugs, AMOEBA focuses on performance differences among semantically equivalent queries, CERT addresses cardinality estimation issues, PUPPY identifies configuration-related problems, and HULK detects data-sensitive anomalies. (2) Methodology: As shown in Table 3, 147 missed optimizations found by SCor trace back to initial releases, indicating their long-standing nature. This makes them undetectable by APOLLO, which relies on cross-version comparison of identical queries. Similarly, AMOEBA may fail when semantically equivalent queries exhibit similarly severe performance degradation. CERT cannot detect opportunities that persist despite correct cardinality estimates. For example, in Listing 2, both queries have accurate cardinality estimates, yet the absence of short-circuit evaluation represents a distinct optimization gap beyond cardinality concerns. PUPPY identifies performance issues by modifying DBMS optimization configurations, which primarily reveals scenarios where existing optimizations perform poorly, but cannot detect missed optimizations that were never implemented.

6 Discussion

SCor Scope. SCor systematically uncovers missed optimizations through unbalanced short-circuit query construction. By leveraging the prevalence of short-circuit patterns across SQL constructs, it exposes various optimization gaps in two key areas. First, it identifies cases where DBMSs fail to apply short-circuit evaluation (e.g., Listing 6). Second, it utilizes short-circuit evaluation to validate other crucial optimizer decisions, such as expression and statement evaluation order (e.g., Listing 2).

SCor Generalizability. SCor is designed as a non-intrusive, black-box testing tool that can detect missed optimizations in DBMSs. This design makes SCor readily applicable to a wide range of DBMSs, regardless of code availability. Although our evaluation primarily targets relational DBMSs, the approach generalizes to other types of systems as well. For example, in graph DBMSs, query languages such as Cypher [32] support constructs like `OR`/`AND` expressions, `WHERE`/`LIMIT` clauses, and `COALESCE()` function. SCor can exploit these features to construct unbalanced short-circuit queries, thereby effectively uncovering performance issues in non-relational DBMSs. Listing 9 illustrates

a performance bug we discovered in Neo4j [30] after generalizing SCor to graph DBMSs. In this case, Neo4j applies short-circuit evaluation when processing `WHERE 1 = 2` (which can be optimized to `WHERE false`), but fails to apply the same optimization for the semantically equivalent predicate `WHERE 1 <> 1`. Neo4j developers confirmed this issue and fixed it.

Listing 9. A performance bug in Neo4j we detected after applying SCor to graph DBMSs, demonstrating the generalizability of SCor.

```
-- The following queries perform a Cartesian product of 1,000 and 500,000 Person nodes.
-- The first query takes 0.001s, as “1 = 2” can be optimized as “false”.
MATCH (p:Person) WITH p LIMIT 1000 MATCH (q:Person WHERE 1 = 2 RETURN p;
-- The second query takes 27.538s, as “1 <> 1” fails be optimized as “false”.
MATCH (p:Person) WITH p LIMIT 1000 MATCH (q:Person) WHERE 1 <> 1 RETURN p;
```

Future Work. SCor enables several promising opportunities for future work. First, the catalog of short-circuit evaluation patterns can be further expanded. Leveraging SCor's platform, researchers can conveniently design and test new patterns, potentially uncovering a wider range of performance issues. Although this work focuses on ten representative short-circuit patterns, identifying entirely new classes could further improve DBMS robustness. Second, the concept of short-circuit evaluation could be extended to detect other categories of bugs beyond performance, such as logic bugs that impact query correctness. Third, the methodology of constructing unbalanced short-circuit queries could be systematically generalized to other DBMS paradigms, such as graph DBMSs, to further validate and broaden the applicability of SCor.

7 Related Work

DBMS Performance Bug Testing. The most related strand of research is on automatic detection of performance issues in DBMSs. APOLLO [18] detects performance regression bugs by comparing the execution time of the same query across different DBMS versions. AMOEBA [24] identifies unexpected slowdowns by contrasting the execution times of semantically equivalent queries. CERT [1] tackles performance bugs through cardinality estimation analysis. MOZI [23] detects performance bugs by leveraging variations in configuration-related configurations. PUPPY [54] detects performance degradation bugs through limited optimization plan construction. HULK [55] detects data-sensitive performance anomalies via data-driven analysis.

As we mentioned in Section 5.4, SCor differs from these testing works in both target and methodology. Overall, existing approaches primarily focus on identifying performance degradations caused by faulty or suboptimal implementations of optimization, whereas SCor is designed to expose optimization opportunities that are entirely missed by the optimizer. Moreover, SCor introduces a novel methodology that turns short-circuit semantics into a performance testing oracle for DBMSs. By constructing unbalanced short-circuit queries whose results depend only on low-cost operations, SCor enables black-box identification of missed optimizations.

DBMS Performance Benchmark. Benchmarking DBMSs is a widely adopted practice for detecting performance regressions and driving continuous performance improvements. Among the various benchmarks available, TPC-H [52] and TPC-DS [51] stand out as the most widely recognized and are regarded as the industry standards. The Join Order Benchmark (JOB) [21] was designed to evaluate query optimizers by testing cardinality estimation, cost models, and plan enumeration techniques on complex, realistic multi-join queries over real-world data. More recently, SQLStorm [45] introduced a new paradigm in DBMS benchmarking by leveraging large language models (LLMs) to generate vast, diverse, and realistic SQL workloads automatically.

SCor complements these benchmarks by addressing an orthogonal problem. While traditional benchmarks primarily detect performance regression bugs using user-relevant workloads, SCor

exposes missed optimization opportunities that standard benchmarks rarely reveal, by constructing unbalanced short-circuit queries, even on previously less-seen workloads.

DBMS Fuzzing. Besides testing DBMS performance, automated testing approaches have also been developed to uncover other types of bugs. The recent application of fuzzing to DBMSs has successfully uncovered hundreds of bugs [19, 22, 42–44, 46, 56]. A typical DBMS fuzzer operates by continuously generating SQL test cases, executing them on the target system, and monitoring for incorrect behavior. These approaches are broadly categorized as either generation-based or mutation-based fuzzing. Generation-based fuzzers construct SQL queries from scratch using predefined syntax or grammar models. For example, SQLSmith [46] generates queries with built-in AST generation rules to expose memory errors. Similarly, SQLancer uses a generation-based approach guided by its test oracles (PQS [44], NoREC [42], TLP [43], and SRS [19]) to detect logic bugs. In contrast, mutation-based fuzzers generate new queries by mutating existing ones. SQUIRREL [56], LEGO [22], and Ratel [53] perform mutations based on a query's AST structure. DynSQL [17] incrementally constructs complex queries by leveraging state information. GRIFFIN [15] adopts a grammar-free approach that reshuffles statements from different queries and restores semantic correctness by tracking the database schema.

SCor could be regarded as a generation-based fuzzer. Different from other generation-based fuzzers, SCor generates SQL queries that incorporate a diverse range of short-circuit patterns, enabling the systematic discovery of missed optimization opportunities.

8 Conclusion

In this paper, we present SCor, a black-box approach for identifying missed optimizations in DBMSs through unbalanced short-circuit query construction. Our key insight is that the results of many queries can be determined without full execution, yet DBMSs still execute the entire query, revealing missed optimizations. SCor realizes it by constructing unbalanced short-circuit queries, whose results can be obtained from low-cost operations alone. If the DBMS still executes the full query with high-cost operations, it indicates missed optimizations. Since short-circuit patterns are prevalent in SQL queries and can be flexibly embedded into diverse query structures, SCor can be systematically applied to expose missed optimizations across a wide range of queries.

We evaluated SCor on 11 mature, widely-used DBMSs, demonstrating its effectiveness and practical value. In total, SCor uncovered 153 previously unknown, unique missed optimization. Of these, 125 have been confirmed by developers, 33 have been fixed, and 28 remain under investigation. These missed optimizations had severe performance implications, had persisted undetected for a long time, and have now been acknowledged by many DBMS developers for fixing. Compared with existing approaches, SCor uniquely identified a large set of non-overlapping bugs. Overall, our work demonstrates that missed optimizations remain a prevalent and overlooked source of missed optimizations, and that SCor offers an effective and targeted method for their detection.

Data Availability

The artifact of SCor is available at https://anonymous.4open.science/r/SCor

Acknowledgements

We sincerely thank the reviewers for their valuable suggestions. This work was supported by the National Natural Science Foundation of China (No. 62572222) and the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM122).

References

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

[2] Stefan Brass and Christian Goldberg. 2006. Semantic errors in SQL queries: A quite complete list. *Journal of Systems and Software* 79, 5 (2006), 630–644. doi:10.1016/j.jss.2005.06.028 Quality Software.

[3] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2002. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Transactions on Database Systems (TODS)* 27, 2 (2002), 153–187.

[4] S. Ceri and G. Gottlob. 1985. Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Transactions on Software Engineering* SE-11, 4 (1985), 324–345. doi:10.1109/TSE.1985.232223

[5] Stefano Ceri and Georg Gottlob. 2006. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on software engineering* 4 (2006), 324–345.

[6] ClickHouse. 2026. https://clickhouse.com/ Accessed: April 19, 2026.

[7] CockroachDB. 2026. https://www.cockroachlabs.com/ Accessed: April 19, 2026.

[8] CockroachDB. 2026. Short-circuit Evaluation in COALESCE() Function. https://www.cockroachlabs.com/docs/stable/functions-and-operators Accessed: April 19, 2026.

[9] CockroachDB. 2026. Short-Circuit in Cross Join with Limit 1. https://github.com/cockroachdb/cockroach/issues/148145 Accessed: April 19, 2026.

[10] DB-Engines. 2026. The DB-Engines Ranking ranks database management systems according to their popularity. https://db-engines.com/en/ranking Accessed: April 19, 2026.

[11] Weimin Du, Ravi Krishnamurthy, and Ming-Chien Shan. 1992. Query Optimization in a Heterogeneous DBMS. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 277–291.

[12] DuckDB. 2026. https://duckdb.org/ Accessed: April 19, 2026.

[13] Andrew Eisenberg and Jim Melton. 1999. SQL: 1999, formerly known as SQL3. *SIGMOD Rec.* 28, 1 (March 1999), 131–138. doi:10.1145/309844.310075

[14] Grant Fritchey. 2012. Execution Plan Cache Analysis. In *SQL Server 2012 Query Performance Tuning*. Springer, 241–279.

[15] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2023. Griffin: Grammar-Free DBMS Fuzzing. 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 49, 12 pages. doi:10.1145/3551349.3560431

[16] Piyush Goel and Bala Iyer. 1996. SQL query optimization: reordering for a general class of queries *(SIGMOD '96)*. Association for Computing Machinery, New York, NY, USA, 47–56. doi:10.1145/233269.233318

[17] Zuming Jiang, Jiaju Bai, and et al. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4949–4965. https://www.usenix.org/conference/usenixsecurity23/presentation/jiang-zu-ming

[18] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: automatic detection and diagnosis of performance regressions in database systems. *Proc. VLDB Endow.* 13, 1 (sep 2019), 57–70. doi:10.14778/3357377.3357382

[19] 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. *Proc. ACM Manag. Data* 3, 6, Article 363 (Dec. 2025), 24 pages. doi:10.1145/3769828

[20] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2021. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering* 6, 1 (2021), 86–101.

[21] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[22] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-Oriented DBMS Fuzzing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 668–681. doi:10.1109/ICDE55515.2023.00057

[23] Jie Liang, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Chengnian Sun, and Yu Jiang. 2024. Mozi: Discovering DBMS Bugs via Configuration-Based Equivalent Transformation. 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 135, 12 pages. doi:10.1145/3597503.3639112

[24] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering*. 225–236.

[25] MariaDB. 2026. https://mariadb.org/ Accessed: April 19, 2026.

[26] MariaDB. 2026. FirstMatch Strategy. https://mariadb.com/docs/server/ha-and-performance/optimization-and-tuning/query-optimizations/optimization-strategies/firstmatch-strategy Accessed: April 19, 2026.

[27] Songsong Mo, Yile Chen, Hao Wang, Gao Cong, and Zhifeng Bao. 2023. Lemo: A Cache-Enhanced Learned Optimizer for Concurrent Queries. 1, 4, Article 247 (Dec. 2023), 26 pages. doi:10.1145/3626734

[28] MySQL. 2026. https://www.mysql.com/ Accessed: April 19, 2026.

[29] MySQL. 2026. Limit Optimization in MySQL. https://dev.mysql.com/doc/refman/8.4/en/limit-optimization.html Accessed: April 19, 2026.

[30] Neo4j. 2026. https://neo4j.com/ Accessed: April 19, 2026.

[31] OceanBase. 2026. https://www.oceanbase.com/ Accessed: April 19, 2026.

[32] The openCypher Implementers Group. 2026. Cypher Query Language Reference. https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf Accessed: April 19, 2026.

[33] OpenGauss. 2026. https://opengauss.org/ Accessed: April 19, 2026.

[34] Oracle. 2026. https://www.oracle.com/database/ Accessed: April 19, 2026.

[35] Oracle. 2026. The CASE Expression in Oracle. https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CASE-Expressions.html Accessed: April 19, 2026.

[36] Oracle. 2026. Short-circuit Evaluation in Logical Operators. https://docs.oracle.com/cd/E19253-01/817-6223/chp-typeopexpr-6/index.html Accessed: April 19, 2026.

[37] PingCAP. 2026. The EXPLAIN statement in TiDB. https://docs.pingcap.com/tidb/stable/sql-statement-explain/ Accessed: April 19, 2026.

[38] Alban Ponse and Daan JC Staudt. 2018. An independent axiomatisation for free short-circuit logic. *Journal of Applied Non-Classical Logics* 28, 1 (2018), 35–71.

[39] PostgreSQL. 2026. https://www.postgresql.org/ Accessed: April 19, 2026.

[40] PostgreSQL. 2026. CREATE TABLE AS SELECT (CTAS) Statements in PostgreSQL. https://www.postgresql.org/docs/current/sql-createtableas.html Accessed: April 19, 2026.

[41] PostgreSQL. 2026. Set Returning Functions. https://www.postgresql.org/docs/current/functions-srf.html Accessed: April 19, 2026.

[42] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1140–1152. doi:10.1145/3368089.3409710

[43] 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

[44] 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

[45] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4144–4157.

[46] Andreas Seltenreich, Bo Tang, and et al. 2015. https://github.com/anse1/sqlsmith.git Accessed: April 19, 2026.

[47] SQLite. 2026. https://sqlite.org/ Accessed: April 19, 2026.

[48] SQLite. 2026. IF() Function in SQLite. https://sqlite.org/lang_corefunc.html Accessed: April 19, 2026.

[49] SQLite. 2026. SQLite Fix Example. https://sqlite.org/src/info/e33da6d5dc964db8 Accessed: April 19, 2026.

[50] TiDB. 2026. https://www.pingcap.com/ Accessed: April 19, 2026.

[51] TPC. 2026. TPC-DS Benchmark. https://www.tpc.org/tpcds/ Accessed: April 19, 2026.

[52] TPC. 2026. TPC-H Benchmark. https://www.tpc.org/tpch/ Accessed: April 19, 2026.

[53] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 328–337. doi:10.1109/ICSE-SEIP52600.2021.00042

[54] Zhiyong Wu, Jie Liang, Jingzhou Fu, Mingzhe Wang, and Yu Jiang. 2024. PUPPY: Finding Performance Degradation Bugs in DBMSs via Limited-Optimization Plan Construction. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 560–571.

[55] Zhiyong Wu, Jie Liang, Jingzhou Fu, Mingzhe Wang, and Yu Jiang. 2025. Hulk: Exploring Data-Sensitive Performance Anomalies in DBMSs via Data-Driven Analysis. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 2181–2202.

[56] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 955–970.

Received October 2025; revised January 2026; accepted February 2026