Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries

Xinyu Liu*, Qi Zhou†, Joy Arulraj*, Alessandro Orso*
*Georgia Institute of Technology, Atlanta, GA, USA; liuxy@gatech.edu, arulraj@gatech.edu, orso@cc.gatech.edu
†Meta, Seattle, WA, USA; zhouqi@fb.com

Abstract

Because modern data-intensive applications rely heavily on database systems (DBMSs), developers extensively test these systems to eliminate bugs that negatively affect functionality. Besides functional bugs, however, there is another important class of faults that negatively affect the response time of a DBMS, known as performance bugs. Despite their potential impact on end-user experience, performance bugs have received considerably less attention than functional bugs. To fill this gap, we present AMOEBA, a technique and tool for automatically detecting performance bugs in DBMSs. The core idea behind AMOEBA is to construct semantically equivalent query pairs, run both queries on the DBMS under test, and compare their response time. If the queries exhibit significantly different response times, that indicates the possible presence of a performance bug in the DBMS. To construct equivalent queries, we propose to use a set of structure and expression mutation rules especially targeted at uncovering performance bugs. We also introduce feedback mechanisms for improving the effectiveness and efficiency of the approach. We evaluate AMOEBA on two widely-used DBMSs, namely PostgreSQL and CockroachDB, with promising results: AMOEBA has so far discovered 39 potential performance bugs, among which developers have already confirmed 6 bugs and fixed 5 bugs.

CCS Concepts

• **Software and its engineering → Maintaining software**; **Software verification and validation**; • **Information systems → Query optimization**.

Keywords

Differential testing, database testing, query optimization

ACM Reference Format:
Xinyu Liu*, Qi Zhou†, Joy Arulraj*, Alessandro Orso*. 2022. Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510093

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510093

1 Introduction

Database management systems (DBMSs) play a critical role in modern data-intensive applications [17, 33]. For this reason, developers extensively test these systems to improve their reliability and accuracy. For instance, they leverage tools such as SQLSMITH [4] and SQLancer [37–39] to discover crash-inducing or logic bugs in DBMSs. However, the same level of scrutiny has not been applied to *performance bugs*—bugs that affect the time taken by the DBMS to process certain queries. Detecting performance bugs is just as crucial as detecting functional bugs, as delayed responses from the DBMS can dramatically affect the user experience [32, 44].

CHALLENGES. To retrieve the results for a given SQL query, the DBMS invokes a pipeline of complex components (*e.g.*, query optimizer, execution engine) [22, 34]. The overall performance of the DBMS may be reduced due to sub-optimal decisions taken by any of these components and the complex interactions among them [8, 9]. Therefore, performance testing on individual components of the DBMS is in general insufficient to detect performance bugs [21, 24, 29, 30]. Another key challenge for detecting performance bugs in DBMS is defining a test oracle that specifies the correct behavior (*i.e.*, response time) of a performant DBMS for a given SQL query.

There are two lines of research that attempt to address this challenge, both focusing on *performance regressions*. One approach uses a pre-determined performance baseline as the oracle [35, 45, 46] and reports a performance bug if there is a significant deviation. While potentially effective in detecting some performance bugs, this approach is human-intensive and error prone, as it is challenging to construct an accurate performance baseline and to account for variability in DBMS performance (to reduce false positives) [28]. Furthermore, this approach relies on a fixed, limited set of queries from standard benchmarks that only cover a subset of the SQL input domain [42].

The second approach leverages differential testing to discover performance regressions [26] by using an oracle to compare the execution time of the same query on two versions of the DBMS. While this technique does not require a developer-provided, pre-determined baseline, it is only able to detect *regressions*, as (1) it requires two versions of the DBMS, with and without the performance bugs, and (2) focuses on structurally simple queries specially tailored for uncovering regressions.

OUR APPROACH. To address the limitations of existing techniques, we present AMOEBA, a new approach for discovering performance bugs in DBMSs. AMOEBA addresses the challenges discussed above along three dimensions. First, it constructs a *performance oracle* by comparing the execution time of *semantically equivalent queries* (*i.e.*, queries that always return the same result) [19, 48]. When

the target DBMS exhibits a significant difference in execution time on a pair of semantically equivalent queries, this may indicate the presence of a performance bug. Second, it constructs queries tailored to the discovery of performance bugs, by supporting complex structures and computationally expensive SQL operators. Further, because of the large space of SQL queries that AMOEBA can explore, we introduce a feedback mechanism that lets it focus on the subset of the query space that is more likely to uncover performance bugs.

Third, it introduces two types of semantic preserving query mutation rules that are also tailored to performance bugs detection: (1) structural mutations, which transform an input query using a set of query rewrite rules derived from the query optimization literature [23], and (2) expression mutations, which modify expressions within an input query without changing their semantics.

To evaluate our technique, we implemented it and applied it to two widely-used DBMSs: CockroachDB and PostgreSQL. Our results are promising, in that AMOEBA found 39 potential performance bugs, among which developers have confirmed 6 bugs and fixed 5 bugs. We also compared AMOEBA against two other sources of equivalent queries that could be used for detecting performance bugs: a manually-written test suite in a widely-used query optimization framework, and the Ternary Logic Partitioning (TLP) approach [38]. Our results show that the equivalent queries generated by AMOEBA are more likely to detect performance bugs.

CONTRIBUTIONS. This paper makes the following contributions:

- A performance bug detection technique with three new aspects:
  - The use of *query equivalence* to generate performance oracles.
  - Two types of query mutations that preserve the semantics of queries: structural mutations and expression mutations.
  - A feedback mechanism that improves the effectiveness and efficiency of the approach.
- An implementation of the technique that is publicly available [12].
- An evaluation of the technique that shows that it can detect real and relevant previously-unknown performance bugs in two widely-used DBMSs.

2 Motivating Example

Figure 1 shows a motivating example that we use to illustrate how *semantically equivalent* queries can be leveraged for detecting performance bugs in DBMSs and show the significant impact performance bugs can have on the end-user experience.

The example consists of a pair of equivalent queries, Q1 (Figure 1a) and Q2 (Figure 1b), that our technique actually generated based on the SCOTT schema [11] and that detected a real performance bug. We also show, in Figure 2, the logical query plans for Q1 and Q2 (*i.e.*, the sequence of logical operations performed when executing the two queries). Although Q1 and Q2 are equivalent, Q1 runs 1,444× slower than Q2 on the same database in CockroachDB [6] (v20.2.0-alpha).

The difference in performance in the two cases is caused by how the *emp* table, which contains 10 million rows, is processed. For Q1, the DBMS ignores that the maximum number of result tuples is 13 and processes the entire *emp* table anyway. For Q2, conversely, the DBMS considers the LIMIT directive, processes the table by row, and stops after fetching the first qualifying 13 entries. As a result, while Q1 takes 13 seconds to execute, Q2 only takes 9 milliseconds.

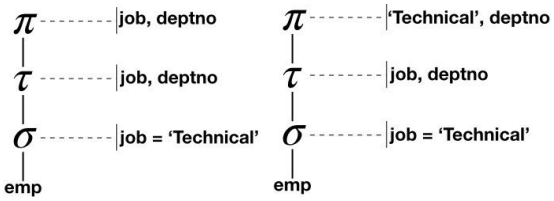
```
SELECT job, deptno FROM emp WHERE  job = 'Technical'
GROUP BY job, deptno LIMIT 13;
```

(a) Original query Q1, execution time = 13 s

```
SELECT CAST('Technical' AS VARCHAR(10)) AS "job", deptno
FROM emp WHERE "job" = 'Technical'
GROUP BY job, deptno LIMIT 13;
```

(b) Mutated query Q2, execution time = 9 ms

Figure 1: Query Rewriting – Example of application of the projection column mutation rule.



(a) Logical query plan for Q1 **(b)** Logical query plan for Q2

Figure 2: Logical Query Plans – sequence of logical operations performed when executing the queries in Figure 1.

The developers have acknowledged that this is a previously-unknown performance bug and have produced a fix for it.

AMOEBA can detect this performance bug because it uses the execution time of equivalent queries as performance oracles. Furthermore, by doing so, AMOEBA can provide a concrete performance bug report, which allows developers to reproduce and investigate the potential performance bug.

3 Background and Terminology

This section provides some relevant background information and introduces the terminology used in the paper.

PERFORMANCE BUGS AND RELATED CONCEPTS. A *performance bug* is a bug that affects the time taken by the DBMS to process certain queries. Before reporting it to the DBMS developers, we refer to a performance discrepancy identified by AMOEBA as a *potential performance bug (PPB)*, which can be *unique* or not, depending on whether its root cause differs from that of other bugs discovered by AMOEBA (based on manual analysis). After we report a PPB, depending on the feedback we receive from the developers, we classify the PPB according to the following taxonomy:

- *Confirmed:* The developers acknowledge that the issue reported indicates an actual performance bug. Confirmed performance bugs can further be classified as either *previously unknown*, if the developers do not refer us to a previous/duplicate bug report or an already planned fix, or *previously known*, otherwise. A confirmed performance bug, whether previously known or not, can also be classified as *fixed*, if the developers plan to fix the it, already fixed it, or have a fix in progress, or *not fixed*, otherwise.
- *Backlogged:* If the developers respond to the report and state that they will analyze the PPB at a later time.
- *Unconfirmed:* If the developers do not acknowledge that the issue reported is a performance bug. This can happen for two reasons. Either the developers disagree that two reported queries are equivalent, in which case we refer to the issue as a *false positive*, or they acknowledge the performance discrepancy but consider it a future/missing optimization, rather than a bug.
- *Unknown:* If the developers ignore the report.

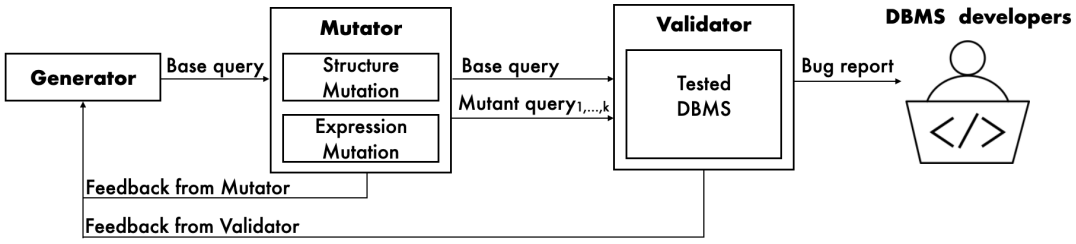


Figure 3: Architecture of AMOEBA– GENERATOR constructs a set of base SQL queries based on feedback from other components. MUTATOR performs semantic-preserving structural and expression mutations on the base queries. VALIDATOR executes a pair of semantically equivalent queries using the target DBMS and reports query pairs that exhibit a significant difference in runtime performance.

SEMANTIC EQUIVALENCE. Two queries are semantically equivalent if they always produce the same result on any input database instance. Query equivalence is a well-studied topic used in many applications, such as testing DBMSs for correctness [38, 40], educating developers [27], and automatically grading student assignments [16]. Unlike prior efforts, we seek to leverage semantic equivalence of queries to find performance bugs in DBMSs.

QUERY REWRITING. AMOEBA constructs equivalent queries by rewriting them using a set of rules that preserve equivalence [23]. For example, a rule could mutate projection columns (*i.e.*, the columns the query should return) using information from the filter predicate (*i.e.*, the predicate that specifies which rows are to be returned). We illustrate how this rule transforms its projection columns while preserving semantic equivalence using query Q1 (Figure 1a) and its logical query plan (Figure 2a). Since the filter clause (σ) in Q1 selects tuples such that the *job* attribute is equal to a specific value, the rule replaces the final projection column *job* with a literal column that takes the same value. Figure 1b and Figure 2b show the transformation result, that is, Q2 and its logical query plan.

4 The AMOEBA Technique

4.1 System Overview

AMOEBA helps developers uncover performance bugs in a DBMS. The key idea is to compare the runtime performance of the DBMS on two semantically equivalent queries, which we would normally expect the DBMS to execute in a similar amount of time. If that is not the case, and the difference in the query execution time exceeds a developer-specified threshold (*e.g.*, 2×), then AMOEBA has found a PPB. Such a *performance oracle* allows us to detect performance bugs in a single DBMS (*i.e.*, without resorting to comparative analysis against another DBMS).

Figure 3 illustrates the architecture of AMOEBA, which contains three main components: (1) GENERATOR, (2) MUTATOR, and (3) VALIDATOR.

❶ GENERATOR leverages a domain-specific fuzzing technique to generate SQL queries *from scratch* based on a database schema. We refer to these queries as *base queries*. GENERATOR is tailored to generate queries that are more likely to trigger performance bugs in DBMSs. In particular, it receives feedback from the latter components of AMOEBA to guide the query generation process.

❷ MUTATOR takes a base query as input and seeks to generate equivalent queries by applying a set of semantics-preserving query-rewriting rules to the query. We refer to the resulting set of equivalent queries as *mutant queries*. The output of this component is the base query and a set of equivalent mutant queries.

❸ VALIDATOR takes a set of equivalent queries as input and generates a list of performance bug reports. It runs each pair of equivalent queries on the target DBMS and observes whether any pair exhibits a significant difference in runtime performance. If so, it first verifies whether this behavior is reproducible across multiple runs. If it can confirm the discrepancy, it generates a report that consists of: (1) the pair of equivalent queries that exhibit the performance discrepancy, and (2) their query execution plans.

We next present the three components of AMOEBA in detail.

4.2 Query Generator

AMOEBA uses a grammar-aware GENERATOR that randomly constructs, given a database schema, a set of base queries from scratch (*i.e.*, without using any seed query). As shown in Algorithm 1, GENERATOR takes a target database as input, generates base queries as output, and uses two key procedures for generating queries that are more likely to trigger performance bugs: (1) GENERATEQUERY (§4.2.1) uses a top-down, grammar-aware approach to generate queries that are compatible with the schema of the input database, and (2) UPDATEPROBTABLEWITHFEEDBACK (§4.2.2) leverages feedback from prior runs of MUTATOR (§4.3) and VALIDATOR (§4.4) to guide the GENERATEQUERY procedure. AMOEBA relies on this feedback mechanism to improve the probability of generating queries that trigger performance bugs. Next, we provide more details about these two procedures.

4.2.1 Grammar-Aware Query Generation. Researchers have extensively explored techniques for grammar-aware query generation [4, 13, 14, 47]. AMOEBA's query generation approach differs from prior work in that it is geared towards generating queries that are *more likely* to trigger performance bugs in DBMS. This part of the approach is based on two main components: (1) a *grammar* for generating queries with different structures and operators, and (2) a *probability table* defined with respect to the grammar to guide the query generation process.

GRAMMAR. AMOEBA uses a grammar based on the SQL-92 standard [1]. The grammar is expressed in Backus–Naur Form (BNF), which consists of both terminal and non-terminal symbols. We show a subset of the grammar in Table 1. For instance, the non-terminal symbol $\text{table}_{\text{ref}}$ may either be a base table (*i.e.*, $\text{table}_{\text{simple}}$) from the target database or a derived table (*i.e.*, $\text{table}_{\text{joined}}$) resulting from a JOIN operator.

Table 1: SQL Grammar – A subset of the SQL grammar that allows AMOEBA to generate queries with a variety of structures and operators.

query_spec	::=	SELECT <column$_{ref}$> FROM <table$_{ref}$> <group_clause> <limit_clause>
table$_{ref}$	::=	<table$_{simple}$> | <table$_{joined}$>
table$_{joined}$	::=	<table$_{ref}$> <join_spec> <table$_{ref}$>
join_spec	::=	<join$_{type}$> ON <join$_{cond}$>
join$_{type}$	::=	LEFT | CROSS | INNER
join$_{cond}$	::=	<bool$_{expr}$> | TRUE

Table 2: Probability Table – Probability values that AMOEBA uses to generate table references and join conditions.

(a) Table References

table$_{simple}$	table$_{joined}$		
	LEFT	CROSS	INNER
0.5	0.16	0.17	0.17

(b) Join Conditions

bool$_{expr}$	TRUE
0.5	0.5

PROBABILITY TABLE. As shown in Table 2, AMOEBA maintains a table that contains the probability of using each non-terminal and terminal symbol when generating queries following the grammar. Thus, this table determines the likelihood of a SQL structure or clause to appear in the generated query. For all symbols that stem from a given non-terminal symbol, the probabilities sum up to one. For instance, Table 2b specifies that there is an equal chance of generating a non-terminal symbol, the JOIN condition, using either a boolean expression (*e.g.*, t1.k = t2.k) or the keyword TRUE.

Next, we present the algorithm of GENERATEQUERY, which is shown in Algorithm 1. *First*, the algorithm acquires information that it needs to generate queries (line 1). Specifically, it performs the following steps: (1) it randomly samples a small dataset from the target database, so as to be able to generate queries with meaningful predicates and a variety of selectivity. (2) it collects table schemas of the target database (*i.e.*, table names, column names, and column types), which it needs to create valid expressions, such as SQL function calls and column comparisons. *Second*, the algorithm initializes with default values the probability table it uses for guiding the query generation procedure (line 2). *Third*, it invokes the BUILDSPECIFICATION function to construct a query specification based on (1) the SQL grammar and (2) the collected meta-data (line 6). *Finally*, the algorithm translates the specification into a well-formed query for the target DBMS (line 7).

4.2.2 Feedback from Mutator and Validator. We now discuss how GENERATOR updates the probability table based on the feedback from MUTATOR and VALIDATOR (line 9).

FEEDBACK FROM MUTATOR. GENERATOR uses the feedback from MUTATOR to improve the likelihood of generating base queries that can be successfully mutated. Since AMOEBA relies on the generation of semantically equivalent queries, this feedback mechanism indirectly increases the likelihood of discovering PPBs. In procedure UPDATEPROBTABLEWITHMUTATORFEEDBACK (line 13), GENERATOR updates the probability table when a base query that it generates is successfully transformed by MUTATOR (line 12). First, it extracts SQL entities from the base query. For example, since Q1 in our motivating example (Section 2) can be successfully mutated into Q2, GENERATOR extracts the following entities from Q1: table$_{simple}$, GROUP BY, and LIMIT. The rationale for this part of the approach is that these entities are correlated with successful mutations. Then,

Algorithm 1: Algorithm for generating SQL queries

```
   Input  : database: database under test
   Output : base queries: random SQL queries
1  meta-data ← RetrieveMetaData(database);
2  prob_table ← InitProbabilityTable();
3  rule_activation_frequency_table ← InitRuleActivationFrequencyTable();
4  Procedure GenerateQuery(meta-data, prob_table)
5      while True do
6          specification ← BuildSpecification(meta-data, prob_table) ;
7          base query ← SpecToQuery(specification, dialect) ;
8          return base query ;
9      prob_table ← UpdateProbTablewithFeedback(base query, prob_table) ;
10
11 Procedure UpdateProbTablewithFeedback(base query, prob_table)
12     if MutateQuery(base query) is True then
           // If the base query can be mutated to generate equivalent
              queries, update the probability table so that it is more
              likely to generate queries containing SQL entities that
              the base query contains.
13         UpdateProbTableWithMutatorFeedback(base query, prob_table,
             applied_rules) ;
14     if TriggerPerformanceBug(base query) is True then
           // If the base query can trigger a performance bug, update
              the probability table so that it is more likely to
              generate queries containing SQL entities that the base
              query contains.
15         UpdateProbTableWithValidatorFeedback(base query, prob_table) ;
16
```

GENERATOR updates the probability table by increasing the values corresponding to these extracted entities and decreasing the values of the other entities. For example, given the entities extracted from Q1, GENERATOR would update the probability table by (1) increasing the probabilities associated with table$_{simple}$, GROUP BY, and LIMIT, and (2) decreasing the probabilities of LEFT, CROSS, and INNERAfter this update, GENERATOR will generate base queries that are more likely to contain the GROUP BY and LIMIT clauses.

GENERATOR also seeks to avoid generating queries that only trigger a limited set of mutation rules, which improves the computational efficiency of AMOEBA (§5.4). To accomplish this, it keeps track of the frequency with which each mutation rule has been fired (line 3). For instance, when a base query triggers a less-frequently triggered mutation rule, GENERATOR increases the probabilities of the entities extracted from that query by a larger amount. In this way, GENERATOR is more likely to construct queries that trigger a wide variety of mutation rules.

FEEDBACK FROM VALIDATOR. GENERATOR utilizes information about discovered PPBs to increase the likelihood of generating base queries that uncover them. When VALIDATOR discovers a base query that eventually leads to finding a performance discrepancy (line 14), GENERATOR invokes the UPDATEPROBTABLEWITHVALIDATORFEEDBACK procedure to suitably update the probability table (line 15). Similar to UPDATEPROBTABLEWITHMUTATORFEEDBACK, this procedure first extracts the SQL entities from the base query, and then updates the probability table accordingly.

4.3 Query Mutator

MUTATOR takes a base query as input and seeks to generate mutant queries that are semantically equivalent to the base query.

QUERY EQUIVALENCE.

MUTATOR leverages and adapts query rewrite rules from the query optimization literature [23] to transform queries while preserving their semantics. We now discuss its mutation rules and algorithm.

Table 3: Illustrative List of Mutation Rules.

Category	Rule Idx	Transformation	Working Example
Structure	13	Push filter through GROUP BY	(t1.c GROUP BY t1.c) WHERE t1.c > 0 → (t1.c WHERE t1.c > 0) GROUP BY t1.C
	59	Propagate LIMIT through LEFT JOIN	(t1 LEFT JOIN t2) SORT t1.c → ((t1 SORT t1.c) LEFT JOIN t2) SORT t1.c
Expression	15	Convert EXTRACT into date range comparison	EXTRACT (YEAR FROM c1) < 2021 → c1 < TIMESTAMP '2021-01-01 00:00:00'
	54	Reduce expression in FILTER	WHERE c1 = CAST(10/2) as INT → WHERE c1 = 5

(a) Rule condition **(b)** Transformation

Figure 4: Illustrative Rule Condition and Transformation (Rule 59).

4.3.1 Mutation Rules. MUTATOR currently supports 75 mutation rules that fall into two categories: (1) rules for mutating query structure (67 rules), (2) rules for mutating SQL expressions (8 rules). Each rule is represented by a triple:

<*id, condition, transformation*> [23].

The *id* is a unique identifier for the rule. The *condition* is a precondition that the original query must satisfy for the rule to be applicable and semantic preserving; it is expressed in terms of the meta-data of the database under consideration, including the data type, PRIMARY KEY, UNIQUE, and NOT NULL information for each column of the database. Finally, the *transformation* is a function that takes the original query as input and returns a mutated query.

Table 3 presents a subset of these rules to illustrate their variety. Due to space limitations, the table does not include the precise *condition* for each rule. A complete list of rules and their implementation are available in our supplementary materials [12].

RULES FOR MUTATING QUERY STRUCTURE. These rules modify the query's structure based on a relational algebraic transformation that preserves equivalence [15]. As an example, Figure 4 illustrates the condition and transformation associated with rule 59 in Table 3. This rule modifies the query structure by propagating the SORT operator below the LEFT JOIN operator. As shown in Figure 4a, this rule is only triggered when the SORT operator is the parent of the LEFT JOIN operator. If the input query satisfies this condition, then this rule transforms it into a structurally-different query shown in Figure 4b by pushing a copy of the SORT operator and its argument below the LEFT JOIN operator.

RULES FOR MUTATING EXPRESSION. These rules rewrite the expression in the query without altering the query structure. Rule 15 in Table 3, for instance, mutates the comparison between a constant value and the result of the EXTRACT function (*e.g.*, EXTRACT (YEAR FROM c1)) < 2021) by replacing the original expression with a new one that compares the attribute against a timestamp.

4.3.2 Mutation Algorithm. We outline the algorithm of MUTATOR in Algorithm 2. MUTATOR takes a base query and the metadata of the target database as input and returns the base query and its semantically equivalent mutant queries as output. It first preprocesses the base query and generates its logical query plan tree, R_{origin}, on which it applies the mutation rules (line 2). It attempts to mutate

Algorithm 2: Procedure for mutating SQL queries

```
   Input  : base query; meta-data: metadata for the target database
   Output : base query and mutant queries
1  Procedure MutateQuery(base query, meta-data)
2      R_origin ← Preprocess(base query) ;
3      transformed_trees ← EmptySet() ;
4      mutant queries ← EmptySet() ;
5      for k ← 1 to number_of_attempts do
           // randomly select a list of mutation rules
6          mutate_rules ← RulesInitialization() ;
7          R_new ← MutateTree(R_origin, mutate_rules, meta_data) ;
8          if R_new ∉ transformed_trees then
9              new query ← TranslateToQuery(R_new, dialect);
10             Update(transformed_trees, mutant queries) ;
11     return base query, mutant queries ;
12 Procedure MutateTree(R_origin, mutate_rules, meta_data)
13     target_expr ← R_origin;
14     for rule ∈ rewrite_rules do
15         target_expr ← ApplyRule(target_expr, rule, meta_data) ;
16     if target_expr ≠ R_origin then
17         return target_expr ;
```

R_{origin} for a total of *number_of_attempts* times (line 5).
In each iteration, MUTATOR performs the following three steps:
❶ It randomly initializes an ordered list of mutation rules (*mutate_rules*), which it then applies on the R_{origin} in a sequential manner (line 6). In doing so, MUTATOR increases the likelihood of uncovering different compositional effects of mutation rules on the input query.
❷ It invokes procedure MUTATETREE to transform R_{origin} using *mutate_rules* (line 7). Within this procedure, MUTATOR uses the database meta-data to check whether the *rule condition* for performing the transformation is met (line 15). Procedure MUTATETREE returns the resulting plan tree, R_{new}, only if R_{new} is different from R_{origin} (line 17).
❸ After getting R_{new} from procedure MUTATETREE, MUTATOR checks whether it is different from trees constructed in prior mutation attempts (line 8). If so, MUTATOR translates R_{new} into a well-formed SQL query, *new query*, based on the target DBMS's dialect and appends it to mutant queries (line 9, line 10). Finally, the algorithm returns the base query and mutant queries as its output.

4.4 Validator

VALIDATOR takes a set of pairs of semantically-equivalent queries as input and generates performance bug reports as output. To do so, it compares the execution time of the queries within each pair. If a pair of equivalent queries consistently exhibit significant difference in their runtime performance, VALIDATOR generates a performance bug report that consists of: (1) the pair of queries, and (2) their execution plans. Before presenting the algorithm of VALIDATOR, we discuss two challenges associated with discovering performance

Algorithm 3: Procedure for detecting performance bugs

```
   Input  : base query                          mutant queries
            validation_attempt: # of additional runs   threshold: least time difference
   Output : bug reports: each report contains a query pair and execution plans
1  if CheckPlanDiff(base query, mutant queries) then
       // Run equivalent queries if they have different execution plans
2      time_list ← RunQuery(base query, mutant queries) ;
3      if Max(time_list) > threshold * Min(time_list) then
           // Rerun queries for validation_attempt times to confirm the
              difference is consistent
4          if Confirm(base query, mutant queries) then
5              GenBugReport(base query, mutant queries) ;
6
7  Procedure CheckPlanDiff(base query, mutant queries)
8      cost_list ← EstimateCost(base query, mutant queries) ;
9      if Count(Set(cost_list)) > 1 then
10         return True ;
11
12 Procedure Confirm(base query, mutant queries, validation_attempts)
13     difference_count ← 0 ;
14     for k ← 1 to validation_attempts do
15         time_list ← RunQuery(base query, mutant queries) ;
16         if Max(time_list) > threshold * Min(time_list) then
17             difference_count += 1 ;
18     if difference_count = validation_attempts then
19         return True ;
20
```

bugs based on query equivalence and how we address them in the algorithm.

EQUIVALENT EXECUTION PLANS. A pair of semantically equivalent queries with different syntax (*i.e.*, structural difference or predicate difference) may reduce to the same query execution plan. In this case, the DBMS will execute these queries in the same way and there will not be any difference in runtime performance. Such queries are not *useful* for discovering performance bugs. Therefore, to improve the computational efficiency of AMOEBA, VALIDATOR focuses on equivalent queries that have different execution plans. In particular, before executing a set of equivalent queries and comparing their runtime performances, it first compares their plans and *skips* this query pair if they have the same plan.

FALSE POSITIVES. The execution time of a query may be affected by system-level factors (*e.g.*, caching behavior of concurrent queries) [24]. To avoid false positives due to these factors, before reporting a PPB to the developers, VALIDATOR verifies that the difference is consistently reproducible by re-executing the same query pair multiple times in isolation and in different execution orders.

VALIDATOR ALGORITHM. Algorithm 3 presents the algorithm of VALIDATOR. The algorithm first invokes procedure CHECKPLANDIFF to filter out equivalent queries that lead to equivalent execution plans (line 1). VALIDATOR assumes that two queries have the same execution plan if their estimated costs are the same. Specifically, given two equivalent queries, it utilizes the EXPLAIN feature of DBMSs (line 8) [41] to compute the estimated cost of each query in the pair. If the estimated plan costs are the same, VALIDATOR considers the two query plans to be equivalent and skips them (line 9).

After discarding pairs of equivalent queries deemed to have identical execution plans, VALIDATOR runs the remaining pairs on the DBMS and records their execution time (line 2). Then, within the resulting set of execution times, VALIDATOR checks whether the ratio of the longest to the shortest query execution time exceeds a developer-specified threshold (line 3). If the ratio exceeds this threshold, VALIDATOR invokes procedure CONFIRM to check whether the runtime performance difference is consistently reproducible (line 4).

Procedure CONFIRM re-executes these queries on the DBMS for multiple runs in random orders and monitors whether the execution time difference still holds (line 14–19). If so, VALIDATOR automatically generates a performance bug report (line 5).

5 Evaluation

To evaluate the effectiveness and generality of AMOEBA, we investigated the following questions:

RQ1. Can AMOEBA find performance bugs in DBMSs? (§5.3)

RQ2. How efficient is AMOEBA? (§5.4)

RQ3. Are all mutation rules created equal with respect to discovering performance bugs? (§5.5)

RQ4. How does AMOEBA compare against other techniques for finding performance bugs? (§5.6)

RQ5. How do the base queries in AMOEBA compare against those in Calcite? (§5.7)

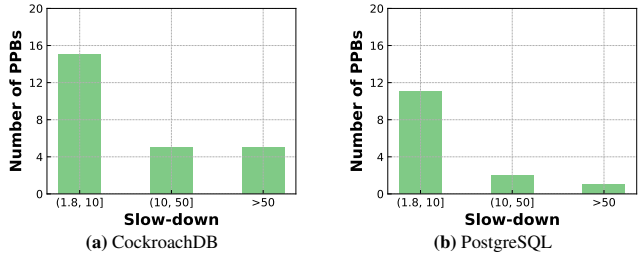
5.1 Implementation

QUERY GENERATION. GENERATOR aims to construct queries that (1) are likely to be syntactically correct and (2) cover a widely-supported subset of SQL constructs [1]. To this end, we implement GENERATOR based on SQLALCHEMY [7], a SQL toolkit and Object Relational Mapper (ORM).

QUERY MUTATION. We build MUTATOR on top of the Calcite [5] query optimization framework. Calcite transforms queries by iteratively applying a set of query rewrite rules [23] and works well with SQLALCHEMY, in that they both cover a widely-supported subset of SQL constructs and dialects. As a consequence, most of the semantically-valid queries constructed by GENERATOR can be processed by the MUTATOR. Furthermore, by leveraging the SQLALCHEMY and Calcite frameworks, AMOEBA can be easily extended.

IMPLEMENTATION SCOPE. AMOEBA supports queries with four data types: integer, double, datetime, and string. The queries may use several SQL constructs (*e.g.*, GROUP, DISTINCT, ORDER and UNION) and functions (*e.g.*, AVG and SUM). We present a detailed list of supported SQL constructs in our supplementary materials [12].

SCHEMA. AMOEBA currently generates queries using the SCOTT schema [11] and runs them on a database based on the same schema. (We made this decision because we seek to compare AMOEBA against the manually-crafted Calcite test suite, which is based on this schema.) It is worth noting that the SCOTT schema is comparatively simple: only contains three tables with two primary keys and one foreign key. We configured the size of the database to 30 MB. Because the query execution time is proportional to the size of the database, we wanted to achieve a balance between discovering reproducible bugs (which requires a larger database) and limiting the computational cost of AMOEBA (which requires a smaller database).



(a) CockroachDB **(b)** PostgreSQL

Figure 5: Runtime Impact of Potential Performance Bugs – Figures (a) and (b) show the slow-down of queries caused by PPBs for CockroachDB and PostgreSQL, respectively.

5.2 Evaluation Setup

Our evaluation focused on two DBMSs: (1) CockroachDB (v20.2.0-alpha), and (2) PostgreSQL (v12.3). We ran all experiments on a server with two Intel(R) Xeon(R) E5649 CPUs (24 processors) and 236 GB RAM. We manually examined the bug reports generated by AMOEBA and reported them to the developers for feedback.

5.3 RQ1 — Performance Bugs Detection

AMOEBA found 25 and 14 PPBs in CockroachDB and PostgreSQL, respectively. Figure 5 summarizes the impact of the discovered performance discrepancies (*i.e.*, the performance gap between equivalent query pairs in the bug report).

RUNTIME IMPACT. While PPBs found in CockroachDB exhibit a slow-down ranging from 1.9× to 669.1×, those found in PostgreSQL exhibit a slow-down between 1.9× to 555.6× for equivalent queries .

DEVELOPERS' FEEDBACK. Overall, developers have confirmed 6 bugs and fixed 5 among those that we reported. However, the reaction was different between the developers of PostgreSQL and CockroachDB. Of the 7 PPBs we reported to them, the PostgreSQL developers considered 4 to be future/missing optimizations that they did not plan to support at the moment, and 1 to be a false positive. Of the remaining 2, they did not respond to 1, and the other 1 matched a planned fix, which should at least indicate that it was considered a critical optimization worth addressing. Notably, the PostgreSQL developer who responded to our reports recommended that we listed the future/missing optimizations that we identified and reported on their official wiki page, which serves as a collaboration area for PostgreSQL developers and users [10]. At the time of this writing, we are in the process of creating and submitting this page.

The CockroachDB developers reacted more positively to our reports, confirming 6 as performance bugs, assigning 18 reports to their backlog, and classifying 1 report as a false positive. Among the confirmed reports, the CockroachDB developers have fixed 5 (acknowledging 2 that matched planned fixes).

In summary, and according to the terminology we introduced in Section 3, AMOEBA identified 1 unconfirmed, previously known, and fixed PPB,[1] 5 unconfirmed PPBs, 1 of which is a false positive, and 1 unknown PPB for PostgreSQL; for CockroachDB, AMOEBA identified 6 confirmed performance bugs (among which 3 were previously unknown and fixed, 2 were previously known and fixed, and 1 was previously unknown and not fixed), 18 backlogged PPBs, and 1 unconfirmed PPB, which is a false positive. We provide details on all the reports submitted and on the developers' reactions to each of them in our supplementary materials [12].

DESCRIPTION OF BUGS. We now discuss a subset of the PPBs found by AMOEBA to illustrate the types of bugs it can find.

EXAMPLE 1: EXPRESSION SIMPLIFICATION. The pair of equivalent queries below exhibit a 3.2× slow-down in CockroachDB.

```
/* [First query, 75 milliseconds] */
SELECT Max(emp.sal)
FROM  dept INNER JOIN emp ON ename NOT LIKE name
WHERE  name = 'ACCT';
/* [Second query, 238 milliseconds] */
SELECT Max(emp.sal)
FROM   dept INNER JOIN emp ON ename NOT LIKE name
WHERE  name = 'ACCT' IS TRUE;
```

The performance difference is caused by the way the filter predicate is processed. For the first query, the DBMS leverages information from the predicate to simplify the JOIN condition, by replacing the variable *name* with the value 'ACCT'. Because of the simplified JOIN condition, the DBMS only needs a partial scan of the table *emp*. Conversely, with the second query, the DBMS decides that it cannot leverage the predicate information to simplify the JOIN condition and scans the entire table *emp*. After analyzing this bug report, the CockroachDB developers realized that a critical predicate normalization rule was missing in their query optimizer. In particular, if an expression within the predicate guarantees to yield a non-null result (*e.g.*, in our example, the non-nullable column *name* compares with a string value), it is safe to reduce operations on top of it that still take null value into consideration [36]. With the second query, this rule would remove the IS TRUE check on top of the comparison clause, which would lead to a more efficient query execution plan. The developers quickly fixed this performance bug due to its broad impact on query performance.

EXAMPLE 2: SUB-QUERIES RETURNING A SCALAR. The pair of equivalent queries below contain predicates that rely on results of the same subquery.

```
/* [First query, 7 milliseconds] */
SELECT sal FROM emp LEFT OUTER JOIN (SELECT job FROM
     bonus LIMIT 1) AS t ON true
WHERE t.job IS NOT DISTINCT FROM 'ACCT';
/* [Second query, 211 milliseconds] */
SELECT sal FROM emp WHERE (SELECT job FROM bonus LIMIT
     1) IS NOT DISTINCT FROM 'ACCT';
```

However, when the predicate is false (*i.e.*, the first tuple of *job* is not 'ACCT'), CockroachDB spends 30× more time to execute the second query compared to the first one. The performance difference stems from how the predicate is processed. For the first query, the DBMS realizes that the predicate in the JOIN operator evaluates to false, and thus skips scanning the *emp* table and executing the JOIN operation. For the second query, however, the DBMS ignores the predicate result and scans the entire *emp* table anyway. The developers quickly confirmed that this performance bug lies in the query execution engine. They also acknowledged that the bug belongs to a more important limitation in the query optimizer, in that it cannot re-optimize the main query based on the results of the sub-query. They plan to fix this issue in the near future.

[1] As we discussed above, this is a somehow peculiar case, as the developers did not confirm that the reported PPB was an actual performance bug, but they had a fix for it in the works.

EXAMPLE 3: HANDLING AGGREGATE OPERATORS. The pair of equivalent queries below trigger a 2.9× execution time difference on PostgreSQL, which exposes a suboptimal behavior when handling an unnecessary GROUP BY operator.

```
/* [First query, 25 milliseconds] */
SELECT emp_pk FROM emp WHERE emp_pk > 100;
/* [Second query, 72 milliseconds] */
SELECT emp_pk FROM emp WHERE emp_pk > 100 GROUP BY
     emp_pk;
```

Specifically, both queries request the DBMS to fetch the *emp_pk* column based on the same predicate. The second query also appends a GROUP BY operation before returning the final table. Since *emp_pk* is the primary key of the *emp* table, the *emp_pk* column takes unique values, thereby rendering the GROUP BY operation unnecessary. For the second query, however, the DBMS performs the GROUP BY operation anyway, which leads to a slower execution time than in the case of the first query. While the developers classified this performance discrepancy as a missing optimization, rather than an actual performance bug, they also mentioned that they were in the process of producing a fix for it. As we mentioned above, this seems to indicate that this performance issue was considered relevant enough to be addressed.

DISCUSSION. The empirical results of applying AMOEBA to test CockroachDB and PostgreSQL show that AMOEBA can effectively detect PPBs. Specifically, the semantically equivalent queries generated by AMOEBA do trigger different runtime behaviors in the DBMSs considered, thereby allowing us to use them as a differential performance oracles for finding PPBs. In addition, we found that the format of our bug reports (*i.e.*, a pair of equivalent queries and their execution plans) seems to provide sufficient information for the DBMS developers to reproduce and investigate the PPB.

> By using the runtime performance of semantically equivalent queries as a performance oracle, AMOEBA discovered 39 PPBs spread across different components of two DBMSs.

5.4 RQ2 — Efficiency

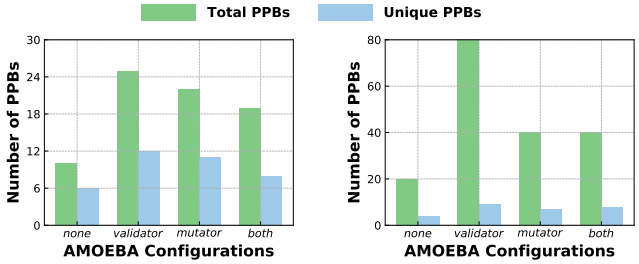
To answer RQ2, we examined the computational efficiency of AMOEBA, as well as whether AMOEBA's feedback mechanisms increase the probability of generating queries that discover PPBs.

In this part of the evaluation, we ran AMOEBA on CockroachDB and PostgreSQL in four different configurations and with a timeout of five hours per configuration:

(1) AMOEBA$_{none}$: both feedback mechanisms disabled,
(2) AMOEBA$_{validator}$: feedback from VALIDATOR enabled,
(3) AMOEBA$_{mutator}$: feedback from MUTATOR enabled, and
(4) AMOEBA$_{both}$ both feedback mechanisms enabled.

Figure 6 presents the results of this study, which consist of the number of total and unique PPBs that AMOEBA discovered in each configuration. We manually mapped each performance bug report to a corresponding unique bug based on the developers' feedback.

BUG REPORTS, UNIQUE BUGS, AND FALSE POSITIVES. We examine the overall efficiency of AMOEBA by averaging the bug-finding results across four runs. As shown in Figure 6, AMOEBA generated an average of 19 performance bug reports (corresponding to 9 unique PPBs) for CockroachDB and an average of 46 PPBs (corresponding to 7 unique PPBs) for PostgreSQL. Finally, among the 182 reports generated for PostgreSQL, 5 are false positives (corresponding to 1 unique PPB).



(a) CockroachDB **(b)** PostgreSQL

Figure 6: Efficiency of AMOEBA– Figures (a) and (b) show the number of total PPBs and unique PPBs that AMOEBA discovers after five hours in CockroachDB and PostgreSQL, respectively.

IMPACT OF FEEDBACK MECHANISMS. To understand the impact of the feedback mechanisms (*i.e.*, feedback from validator and feedback from mutator) on the effectiveness of AMOEBA, we compared the total and unique number of PPBs that AMOEBA discovered in the four different configurations considered. As Figure 6 shows, for both DBMSs, the feedback from validator and mutator increased both the total and unique performance discrepancies that AMOEBA discovered. On CockroachDB, while AMOEBA$_{none}$ only discovered 10 total performance discrepancies and 6 unique performance discrepancies, the other configurations (*i.e.*, AMOEBA$_{validator}$, AMOEBA$_{mutator}$, and AMOEBA$_{both}$) discovered 25, 22, and 19 total performance discrepancies, respectively, which correspond to 12, 11, and 8 unique performance discrepancies. On PostgreSQL, while AMOEBA$_{none}$ only discovered 20 total performance discrepancies and 4 unique performance discrepancies, the other configurations discovered 82, 40, and 40 total performance discrepancies, respectively, which correspond to 9, 7, and 8 unique performance discrepancies.

Based on these results, validator-only feedback seems to be more effective than a combination of validator-and-mutator feedback (especially for CockroachDB). One possible explanation is that leveraging both feedback mechanisms tends to reward a larger number of features and result in more complex queries, which can make it challenging for the DBMS to significantly optimize either query in a pair.

RUNTIME PERFORMANCE. We also counted the number of semantically equivalent queries that AMOEBA examined across four runs. On average, AMOEBA generated and examined a pair of semantically equivalent queries per 1.5 and 0.8 seconds for CockroachDB and PostgreSQL, respectively. The overall runtime performance of AMOEBA was dominated by the runtime of the tested DBMS. To further improve the runtime performance of AMOEBA, it would be possible to deploy it on multiple servers to parallelize its execution.

> AMOEBA detects a large number of PPBs in both CockroachDB and PostgreSQL within the given time limit of 5 hours. On both DBMSs, feedback from validator and mutator both had a significant impact on the number of PPBs discovered.

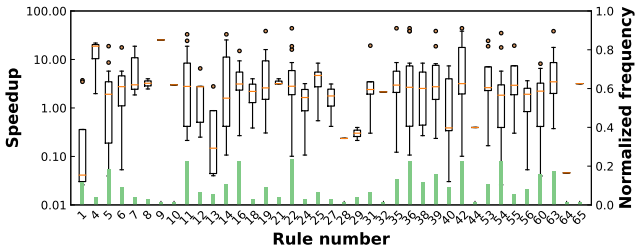


Figure 7: Impact of rules on query performance and frequency of bug-revealing query pairs.

5.5 RQ3 — Effect of Individual Mutation Rules

To answer RQ3, we performed an in-depth analysis of mutation rules used by AMOEBA and their importance in discovering PPBs using the performance bug reports presented in §5.4. Specifically, we examined a dataset of 76 and 182 query pairs that triggered PPBs in CockroachDB and PostgreSQL, respectively. We investigates each mutation rule along two dimensions:

Impact on query performance: If a rule generated a query pair that exhibited a significant runtime performance difference, we measured the speed-up (ratio > 1) or slow-down (ratio < 1).

Frequency of generating bug-revealing query pairs: We counted the number of times each rule generated a query pair that triggered a PPB, normalized by the total number of bug-revealing pairs.

Figure 7 presents the results for both measures for CockroachDB. The impact on query performance is shown using a box plot and the y-axis on the left. The frequency of bug-revealing query pairs is shown using a bar chart and the y-axis on the right. The results for PostgreSQL are analogous.

As the figure shows, not all mutation rules are equally useful in discovering PPBs. Among 75 mutation rules that AMOEBA uses for generating query pairs, only 39 and 44 rules can generate query pairs that trigger PPBs in CockroachDB and PostgreSQL, respectively. These subsets include both structure and expression mutation rules.

With respect to impact on performance, we found that while some rules always had the same effect on query performance (*i.e.*, either speed-up or slow-down), others exhibited different effects (*i.e.*, both speed-up and slow-down) in different cases. Since AMOEBA seeks to mutate each query by applying a sequence of rules, these patterns result from the compositional effects of mutation rules (§4.3.2): (1) *contention* between mutation rules, that is the performance penalty caused by one rule is damped by the gains from another rule, and vice versa; (2) *enabling effect* of mutation rules, that is a rule itself may not affect the query performance but may transform the query into a form that makes it suitable for mutation by other rules (with an effect on performance).

We further studied our results to identify the characteristics of the mutation rules that generate query pairs that can reveal PPBs, as these may indicate query characteristics that make them challenging for a DBMS to optimize and execute. For both DBMSs, we found that these "effective" rules mostly re-arrange or eliminate expensive operators (UNION, GROUP BY, and JOIN) in a given query. Conversely, other rules that manipulate the projection and sorting operators (SELECT and ORDER BY) are less likely to generate query pairs that trigger PPBs. Improving how the DBMS handles these operators would therefore enhance their robustness.

Table 4: Comparative Analysis of AMOEBA – Number of PPBs discovered by the set of query pairs in each baseline and by AMOEBA.

Benchmark	PPBs Found	
	Cockroach	PostgreSQL
Benchmark 1 (TLP)	1	1
Benchmark 2 (Calcite)	4	4
Benchmark 3 (Calcite+AMOEBA)	4	6
AMOEBA	25	14

Both structural and expression mutation rules generate queries that can reveal PPBs. Rules differ in their impact on query performance, and some rules that do not directly affect query performance enable the application of other rules. Rules that transform expensive operators seem to be effective in generating queries that trigger PPBs, which highlights opportunities for improving future versions of the tested DBMSs.

5.6 RQ4 — Comparative Analysis

Query equivalence has been leveraged before in related work [16, 27, 40], albeit not to uncover performance bugs. To answer Q4, we compare AMOEBA to three baseline based on two of these existing approaches: Calcite [19, 48] and TLP [38].

BASELINE 1: QUERY PAIRS FROM TLP. TLP [38] construct equivalent queries to discover *logic bugs* in DBMSs. It is based on the observation that any predicate in SQL evaluates to TRUE, FALSE, or NULL [38]. Accordingly, TLP constructs a mutant query that is equivalent to the base query by (1) dividing the base query into three partition queries, wherein each predicate is constructed based on the value of the overall base query's predicate and (2) concatenating these partition queries using the UNION operator. Our first baseline consists of 2000 pairs of equivlent queries generated using TLP.

BASELINE 2: QUERY PAIRS FROM CALCITE. The Calcite test suite [19, 48] consists of tests manually crafted to ensure the correctness of Calcite's query transformation rules; each test transforms an input query using a set of transformation rules and examines whether the resulting query is correct. Our second baseline consists of 373 pairs of equivalent queries from the Calcite test suite, in which we use the input query as the base query and the transformed query as the mutant query. This is an appropriate and challenging baseline because (1) the input queries are manually created to cover a wide range of SQL operators, and (2) the mutant queries are generated using the same set of transformation rules as AMOEBA.

BASELINE 3: QUERY PAIRS FROM CALCITE USING AMOEBA MUTATOR. Our final baseline consists of 373 pairs of equivalent queries generated by applying AMOEBA's MUTATOR on the base queries from the Calcite test suite.

COMPARISON. Once selected these three baselines, we ran each set of baseline query pairs on our SCOTT-based database and compared the number of PPBs discovered by each baseline to those discovered by AMOEBA. Our results are shown in Table 4. As the figure shows, AMOEBA discovered significantly more PPBs than the baselines in both DBMSs. We next analyze the factors that contribute to the efficacy of AMOEBA.

MUTATION RULES AND ALGORITHM. TLP uses a different set of mutation rules than AMOEBA for generating equivalent queries, which were not designed to detect performance bugs. The queries mutated by TLP consistently take longer to execute than their corresponding base query, with an average slow-down of 17×. This happens because the mutated queries force the DBMS to perform additional operations (*i.e.*, fetching the tuples for each partition query and combining the partial results). Given this inherent overhead, TLP is limited in the kinds and number of PPBs it can find.

While Calcite and AMOEBA use the same set of mutation rules, they differ in the way they leverage these rules. Calcite's tests transform the base queries using a small set of mutation rules applied in a specific order. Conversely, AMOEBA's automated mutation strategy exploits all available mutation rules and their compositional effects to generate equivalent query pairs, which increases the chances of generating queries that can discover PPBs (§4.3.2). The effect of this different approach can be observed in the second and third rows of Table 4, for PostgreSQL, where AMOEBA's mutation strategy discovers two more PPBs than the manual mutation in Calcite.

For both DBMSs considered, AMOEBA discovered considerably more PPBs than the baselines based on Calcite and TLP. Two reasons for this better performance are that AMOEBA (1) uses a rule mutation approach tailored to discovering PPBs and (2) performs a broad exploration of the equivalent query space by leveraging all available rules and their compositional effects.

5.7 RQ5 — Analysis of Base Queries

As shown in Table 4, the base queries derived from the Calcite test suite allowed for discovering fewer PPBs than those generated by AMOEBA. To understand why, we compared a set of 2000 base queries generated by AMOEBA to the 373 base queries in Calcite.

SINGLE-CLAUSE ANALYSIS. We first examined the SQL single-clause and type coverage in the two sets. We found that the base queries for AMOEBA and Calcite cover almost the same set of SQL types and operators (*e.g.*, join operators and common keywords). Since Calcite's base queries are less effective than AMOEBA's, we inferred that targeting high single-clause coverage when generating queries is not sufficient for detecting PPBs. We then investigated the importance of considering combinations of different SQL clauses.

TWO-CLAUSE COMBINATION ANALYSIS. We examined two-clause combination coverage for both sets of base queries (*e.g.*, existence of base queries with both GROUP BY and LEFT JOIN). To do this, we constructed a co-occurrence matrix using the base query dataset and SQL clauses supported by both AMOEBA and Calcite. For each base query dataset, we counted the frequency of queries with two SQL clauses, normalized the count by the number of total queries, and plotted the result in a heatmap, shown in Figure 8. Additionally, we identified clause-pair combinations within base queries that triggered performance bugs (§5.4), and highlighted those combinations that are only found by AMOEBA using a ∗ marker in the heatmap. We only present the heatmap for CockroachDB, as the one for PostgreSQL is similar. Because AMOEBA-generated base queries lead to the discovery of more performance discrepancies, we believe these highlighted combinations may represent *interesting* two-clause combinations that are more likely to trigger PPBs.

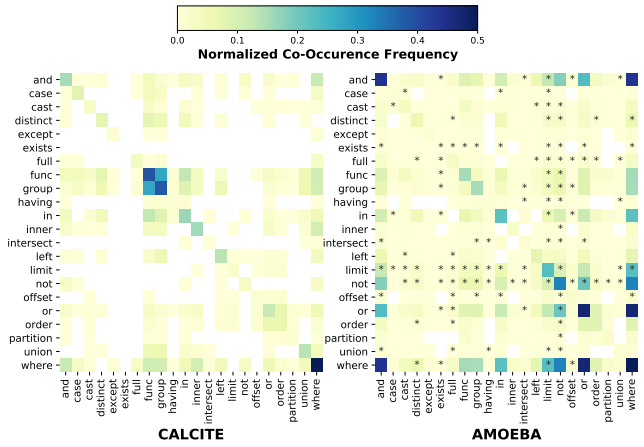


Figure 8: Two-Clause Combinations – AMOEBA-generated base queries cover more clause pair combinations and *interesting* patterns (highlighted with the ∗ marker) than Calcite.

The results in the heatmap show that, for both DBMSs considered, the base queries generated by AMOEBA explore a significantly larger space of two-clause combinations than Calcite's base queries. While base queries from Calcite's tests cover 103 and 106 clause pair combinations for CockroachDB and PostgreSQL, base queries from AMOEBA cover 208 and 200 combinations for the two DBMSs.

POSSIBLE IMPLICATIONS FOR DBMS DEVELOPERS. Our results also show that the base queries from Calcite missed a significant amount of *interesting* clause-pair combinations that lead to discover PPBs. Specifically, they missed 58 and 52 clause pair combinations for CockroachDB and PostgreSQL. We summarize the characteristics of these clause pairs, as they may reflect query patterns that are challenging for DBMS to optimize but are neglected by manual testing efforts: (1) filter clauses (*i.e.*, WHERE and HAVING), when combined with expensive operators such as JOIN, GROUP BY, and UNION, can have a significant effect on query execution time; (2) the LIMIT clause can also affect query execution time. Because LIMIT requests a smaller set of results, an optimal plan should either scan a partial table or terminate expensive operations early. Improving how the DBMSs handle these operations and their combinations may enhance the robustness of its performance.

An additional reason why AMOEBA outperforms the baselines considered, in addition to those discussed in §5.7, is that it covers a wider range of clause-pair combinations that may be challenging for the DBMS to optimize.

6 Limitations

In this section, we discuss the main limitations of AMOEBA and present possible ways to address and mitigate them in future work.

RELIANCE ON AN EXISTING OPTIMIZATION FRAMEWORK. Because AMOEBA is based on the widely-used query optimization framework Calcite, it inherits Calcite's limitations. First, Calcite only focuses on a widely-supported subset of SQL operators and functions, so AMOEBA focuses on the same subset. However, since Calcite is an extensible framework, it is feasible to add support

for additional SQL features. Second, AMOEBA leverages rewrite rules from Calcite. Although these rules are designed to preserve query semantics [23], AMOEBA could generate false positives if for some reason Calcite's rules failed to satisfy this property. To mitigate this threat, AMOEBA generates a bug report only if the two (supposedly) equivalent queries return the same set of output tables, as it is unlikely that queries that are not equivalent would generate the same results.

PERFORMANCE BUGS VS MISSING OPTIMIZATIONS. While AMOEBA can identify relevant cases that developers should consider, it is ultimately the developers' decision whether to support a given optimization. In fact, as discussed in §5.3, the PostgreSQL developers considered most of the the reported issues missing optimizations that they decided to ignore. In some cases, this happens because finding the optimal execution plan is an NP-hard problem [25], so a DBMS may select a sub-optimal plan to reduce computational cost. In other cases, it is simply a design decision. Nevertheless, we believe it is useful to have an automated tool that can report to developers cases that may need consideration and assists them in pinpointing the root cause of a performance discrepancy.

Related to this point, further analysis of the responses from the PostgreSQL developers made us realize that a possible limitation of this approach is that the automated query generation and mutation may result in queries that look artificial and may therefore alienate the developers rather than compel them to investigate and fix the corresponding issues. To address this problem, in future work, we will investigate ways to simplify the automatically generated query pairs and make them more representative of queries that can be encountered in practice.

Finally, it is also worth noting that a byproduct of this work is that it shows that SQL is not fulfilling one of its key promises—that developers can write queries in any form and leave optimization to the DBMS.

7 Related Work

In this section, we present prior work on testing DBMSs with an emphasis on DBMS performance.

FUZZING DBMSS. Given the large state space of possible SQL queries, fuzzing has been applied to find crash bugs and security vulnerabilities in DBMSs [2–4]. Researchers have improved the efficacy of the fuzzing loop by taking the feedback from the tested DBMS into consideration [13, 47]. While AMOEBA is also a fuzzing tool equipped with a feedback mechanism, it differs from prior work in that it focuses on generating semantically equivalent query pairs that trigger different runtime performance.

DIFFERENTIAL AND METAMORPHIC TESTING. To circumvent the oracle problem associated with automated testing, researchers have applied *differential* and *metamorphic* testing techniques for discovering logic bugs in DBMSs [18, 31]. RAGS discovers logic bugs by executing the same query on different DBMSs and comparing the results [42]. Waas *et al*. propose a framework for validating the query optimizer by executing alternative execution plans for the input query and comparing their results [43]. TLP is the state-of-the-art tool for discovering logic bugs in DBMS using metamorphic testing [38]. However, as discussed in §5.6, TLP is not suitable for discovering performance bugs. Unlike this previous work, AMOEBA is a metamorphic testing technique tailored for discovering performance bugs in DBMS.

PERFORMANCE TESTING. Researchers have presented techniques for finding performance bugs by executing the DBMS on pre-defined workloads and comparing their behavior against performance baselines [26, 35, 45, 46]. These techniques detect performance regressions caused by DBMS upgrades and configuration changes. AMOEBA differs from these approaches in that it does not require a pre-defined baseline for finding performance bugs. Instead, it leverages the tested DBMS's runtime behaviors on equivalent queries as a performance oracle.

OPTIMIZER TESTING. Researchers have proposed techniques for testing the query optimizer's ability to find the best execution plan [21, 24]. Li *et al*. propose a benchmark for assessing the efficiency of a query optimizer (*i.e*., optimization time) [30]. Leis *et al*. investigate the impact of the components of the query optimizer on runtime performance [29]. These efforts are geared towards quantifying the quality of an optimizer. Another line of research focuses on developing frameworks for testing the correctness of query transformation rules in the query optimizer [20, 43]. This work requires an in-depth knowledge about the tested query optimizer. AMOEBA complements these efforts by taking a black-box approach and facilitates a more extensive testing of optimizers.

Acknowledgments

This work was partially supported by NSF, under grants CCF-1563991, CCF-0725202, IIS-1850342, and IIS-1908984, DARPA, under contract N66001-21-C-4024, ONR, under contract N00014-18-1-2662, DOE, under contract DE-FOA-0002460, Adobe, the Alibaba Innovative Research Program, Cisco, Facebook, Google, IBM Research, Intel, and Microsoft Research. We thank the developers at CockroachDB and PostgreSQL for their useful feedback on our bug reports.

8 Conclusion

We presented AMOEBA, a new approach for detecting performance bugs in DBMSs. The key idea behind AMOEBA is to construct two semantically equivalent queries and then compare the time it takes the DBMS under test to execute the two queries. If the execution time for the two queries is significantly different, that indicates a potential performance bug in the DBMS. In order to boost the effectiveness and efficiency of AMOEBA, we also defined a query generation and two feedback mechanisms that allow it to focus on the subset of the query space that is more likely to uncover performance bugs. To assess our approach, we implemented AMOEBA and evaluated it on two widely-used DBMSs with encouraging results. AMOEBA was able to discover 39 potential performance bugs. Developers already confirmed 6 of these bugs and fixed 5 of them.

In future work, we plan to apply AMOEBA to additional DBMSs and to improve our approach based on our current and future findings. Our current results, for instance, highlight relevant query patterns that DBMS may have difficulties processing efficiently. We can use this information to improve AMOEBA by adding to it more rules that focus on such patterns. We will also investigate debugging techniques that can help DBMS developers investigate the root cause of a performance bug after it has been reported.

References

[1] 1992. Database Language SQL. http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

[2] 2015. AFL: American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[3] 2016. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz.

[4] 2016. SQLSmith. https://github.com/anse1/sqlsmith.

[5] 2020. Apache Calcite. https://calcite.apache.org/.

[6] 2020. CockroachDB. https://www.cockroachlabs.com/docs/releases/v20.2.0.

[7] 2020. SQLAlchemy. https://www.sqlalchemy.org/.

[8] 2021. CockroachDB Performance Bug Reports. https://github.com/cockroachdb/cockroach/issues?q=performance.

[9] 2021. PostgreSQL Performance Bug Reports. https://www.postgresql.org/search/?m=1&q=performance&l=8&d=-1&s=r.

[10] 2021. PostgreSQL Wiki. https://wiki.postgresql.org/wiki/MainPage.

[11] 2021. SCOTT schema. https://www.orafaq.com/wiki/SCOTT.

[12] 2022. Supplementary material. https://bit.ly/3I995jL

[13] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. 2007. A genetic approach for random testing of database systems. In *VLDB*. 1243–1251.

[14] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. 2006. Generating Queries with Cardinality Constraints for DBMS Testing. In *TKDE*. 1721–1725.

[15] Stefano Ceri and Georg Gottlob. 1985. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. In *TSE*. 324–345.

[16] Bikash Chandra, Bhupesh Chawda, Biplab Kar, KV Maheshwara Reddy, Shetal Shah, and S Sudarshan. 2015. Data generation for testing and grading SQL queries. In *VLDB Journal*. 731–755.

[17] CL Philip Chen and Chun-Yang Zhang. 2014. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. In *Information sciences*. 314–347.

[18] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. In *arXiv preprint arXiv:2002.12543*.

[19] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL.. In *CIDR*.

[20] Hicham G Elmongui, Vivek Narasayya, and Ravishankar Ramamurthy. 2009. A framework for testing query transformation rules. In *SIGMOD*. 257–268.

[21] Leo Giakoumakis and César A Galindo-Legaria. 2008. Testing SQL Server's Query Optimizer: Challenges, Techniques and Experiences. In *Data Engineering Bulletin*. 36–43.

[22] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. In *CSUR*. 73–169.

[23] Goetz Graefe. 1995. The cascades framework for query optimization. In *Data Engineering Bulletin*. 19–29.

[24] Zhongxian Gu, Mohamed A. Soliman, and Florian M. Waas. 2015. Testing the accuracy of query optimizers. In *DBTest*. 1–6.

[25] Toshihide Ibaraki and Tiko Kameda. 1984. On the optimal nesting order for computing n-relational joins. In *TODS*. 482–502.

[26] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. Apollo: Automatic detection and diagnosis of performance regressions in database systems. In *VLDB*. 57–70.

[27] R Kearns, Stephen Shead, and Alan Fekete. 1997. A teaching system for SQL. In *ACSE*. 224–231.

[28] D. Lee, S. K. Cha, and A. H. Lee. 2012. A Performance Anomaly Detection and Analysis Framework for DBMS Development. In *TKDE*. 1345–1360.

[29] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really?. In *VLDB*. 204–215.

[30] Zhan Li, Olga Papaemmanouil, and Mitch Cherniack. 2016. OptMark: A Toolkit for Benchmarking Query Optimizers. In *CIKM*. 2155–2160.

[31] William M McKeeman. 1998. Differential testing for software. In *Digital Technical Journal*. 100–107.

[32] Rasha Osman and William J. Knottenbelt. 2012. Database System Performance Evaluation Models: A Survey. In *Performance Evaluation*. 471–493.

[33] Jaroslav Pokornỳ. 2015. Database technologies in the world of big data. In *CompSysTech*. 1–12.

[34] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database Management Systems*.

[35] Kim-Thomas Rehmann, Changyun Seo, Dongwon Hwang, Binh Than Truong, Alexander Boehm, and Dong Hun Lee. 2016. Performance Monitoring in SAP HANA's Continuous Integration Process. In *PER*. 43–52.

[36] Raymond Reiter. 1986. A Sound and Sometimes Complete Query Evaluation Algorithm for Relational Databases with Null Values. In *J. ACM*. 349–370.

[37] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *FSE*.

[38] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. In *OOPSLA*.

[39] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *OSDI*.

[40] Shetal Shah, S Sudarshan, Suhas Kajbaje, Sandeep Patidar, Bhanu Pratap Gupta, and Devang Vira. 2011. Generating test data for killing SQL mutants: A constraint-based approach. In *ICDE*. 1175–1186.

[41] Hitesh Kumar Sharma, Mr SC Nelson, et al. 2017. Explain Plan and SQL Trace the Two Approaches for RDBMS Tuning. In *Database Systems Journal*. 31–39.

[42] Donald R Slutz. 1998. Massive Stochastic Testing of SQL. In *VLDB*. 618–622.

[43] Florian Waas and César Galindo-Legaria. 2000. Counting, Enumerating, and Sampling of Execution Plans in a Cost-Based Query Optimizer. In *SIGMOD*. 499–509.

[44] Gerhard Weikum, Axel Moenkeberg, Christof Hasse, and Peter Zabback. 2002. Self-Tuning Database Technology and Information Services: From Wishful Thinking to Viable Engineering. In *VLDB*. 20–31.

[45] Khaled Yagoub, Peter Belknap, Benoit Dageville, Karl Dias, Shantanu Joshi, and Hailing Yu. 2008. Oracle's SQL Performance Analyzer.. In *Data Engineering Bulletin*. 51–58.

[46] Jiaqi Yan, Qiuye Jin, Shrainik Jain, Stratis D Viglas, and Allison Lee. 2018. Snowtrail: Testing with Production Queries on a Cloud Database. In *DBTEST*.

[47] 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 *CCS*. 955–970.

[48] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated verification of query equivalence using satisfiability modulo theories. In *VLDB*. 1276–1288.