PINOLO: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis

Zongyin Hao[1], Quanfeng Huang[1], Chengpeng Wang[2], Jianfeng Wang[3], Yushan Zhang[4], Rongxin Wu[1]*, Charles Zhang[2]

[1]School of Informatics, Xiamen University, [2]The Hong Kong University of Science and Technology, [3]University of Southern California, [4]Tencent Inc.

{haozongyin, huangquanfeng}@stu.xmu.edu.cn, {cwangch, charlesz}@cse.ust.hk, jianfenw@usc.edu, wurongxin@xmu.edu.cn

Abstract

DBMSs (Database Management Systems) are essential in modern enterprise software. Thus, ensuring the correctness of DBMSs is critical for enterprise applications. Among various kinds of bugs, logical bugs, which make a DBMS return an incorrect result set for a given SQL query, are the most challenging for detection since they typically do not result in apparent manifestations (e.g., crashes) and are likely to go unnoticed by users. The key challenge of detecting logical bugs is the test oracle problem, i.e., how to automatically characterize the expected results for a given query. The state-of-the-art approaches focus on generating the equivalent forms of queries via the customized rules, which rewrite a seed query to achieve the equivalent transformation. This dramatically limits the forms of SQL queries fed to the DBMS and thus leads to the under-reporting of many deeply-hidden logical bugs. In this paper, we propose a novel approach, PINOLO, to constructing a test oracle for logical bugs. Instead of generating the equivalent mutants of a seed query, our idea is to synthesize the queries that theoretically should return a superset or a subset of the result set of the seed query, forming the over-approximations or under-approximations of the seed query. A logical bug is detected if the result set returned by our synthesized query does not follow the expected approximation relation. We implemented our idea as a DBMS testing system and evaluated it on four widely-used DBMSs: MySQL, MariaDB, TiDB, and OceanBase. By the time of writing, PINOLO has found 41 unique logical bugs in these DBMSs, 39 of which have been confirmed by developers.

1 Introduction

Database Management Systems (DBMSs) are widely used as a key component in modern enterprise software. Their correctness and reliability are critical for many enterprise applications, such as online banking, e-shopping, e-payment, etc. Therefore, DBMS testing has attracted considerable attention in the industry [14, 23, 38, 41] and academia [10, 34–36]. For example, fuzzing, a widely-used testing technique, has been extensively applied to DBMSs [14, 38], showing its effectiveness in detecting crash bugs. However, as another typical kind of bug, logical bugs would cause DBMSs to return an incorrect result set for a given query but can easily go unnoticed by developers since they would not behave with apparent manifestations like system crash.

The predominant approach to detecting logical bugs consists of various automatic testing techniques. However, designing effective automatic testing techniques is non-trivial. One of the fundamental technical challenges is to characterize a correct result concerning a given query for comparison, which is a classical problem in testing, i.e., *test oracle problem* [12]. To tackle this challenge, researchers have proposed various ways to obtain the test oracle. The first category is based on differential testing [39]. It provides the same generated SQL query to multiple DBMSs for execution and resorts to the querying results to construct the test oracle. More concretely, the inconsistency among the result sets returned by different DBMSs indicates the presence of a potential logical bug. However, as pointed out by the existing studies [34, 36, 39], differential testing cannot be applied when a generated SQL query cannot comply with the grammar of all selected DBMSs or contains operations that have different semantics between different DBMSs. Although all the DBMSs support the common core syntax of SQL, each of them provides various extensions and forms its own dialect [39], which dramatically limits the generality of differential testing.

The second category is the oracle-guided synthesis approach [36], which does not rely on multiple DBMSs and thus mitigates the limitation of differential testing. It first specifies a randomly-selected row in a database table, namely, *pivot row*, as the test oracle and then synthesizes the query whose result set should contain this pivot row. The failure of fetching the row with the synthesized query evidences a potential bug underlying the tested DBMS. However, since such an approach considers only one row each time and the synthesis merely focuses on the *where* clause generation, it would miss logical bugs in various scenarios [34, 35]. For example, those rows that are duplicated to the pivot row are wrongly fetched or omitted, or the values processed by performing operators on the original row data are mistakenly computed and returned. Moreover, as pointed out by some recent studies [34, 35], the synthesis requires domain knowledge of the database dialect's supported operators and functions, and thus the implementation effort is high.

*Corresponding author: Rongxin Wu (wurongxin@xmu.edu.cn)

The third category is the metamorphic testing based approach [34, 35]. It first transforms a given query q into another query q' such that their querying results satisfy a specific relation, which is referred to as a *metamorphic relation*. The violation of the metamorphic relation upon the querying results indicates the wrong result of evaluating q or q'. For example, TLP [35] decomposes a query q into three partitioning sub-queries, each of which computes the result sets for a boolean predicate to be evaluated as **TRUE**, **FALSE**, and **NULL**, respectively, and then constructs an equivalent query q' by performing the union operation on these three sub-queries. NoREC [34] transforms an optimized version of a query into a non-optimized one by the customized rule, e.g., changing "**SELECT** * **FROM** t **WHERE** p" into "**SELECT** (p **IS TRUE**) **FROM** t." Compared with the aforementioned two categories of approaches, metamorphic testing based approaches are much more lightweight to implement and have been proven to be more effective in detecting logical bugs [34, 35]. However, existing studies instantiate the metamorphic relations as equivalent relations, which are still insufficient to detect many deeply-hidden bugs. This is because it is highly possible that, owing to the limited search space of mutations, the pair of equivalent queries would still share common buggy operators and functions and eventually return the same results. In such a case, the oracle from the equivalent query cannot provide any hint to detect logical bugs.

In this work, we present a new metamorphic testing based approach, named PINOLO, to detect logical bugs. Our idea to instantiate the metamorphic relation originates from the observation that the querying result of a given query is essentially a multi-set of tuples. The inclusion relation between the multi-sets, which is the foundation of set theory, is a good choice to characterize the metamorphic relation of two queries. Therefore, we try to mutate a given seed query to obtain the queries over or under-approximating it, of which the querying results are the superset or the subset of the one of the seed query. Based on the approximation relations, we can reveal a logical bug if the actual results violate the approximation relation. To systematically synthesize the two kinds of mutants, we introduce a series of approximate mutators, e.g., strengthening or weakening the predicates in *where* clauses, and propose an approximate query synthesis algorithm to generate the queries that have the over and under-approximation relations with the seed queries. Benefiting from our approximation relation and flexible approximate mutators, PINOLO can seize more opportunities to reveal logical bugs, as it can perform more aggressive mutations over the seed queries (e.g., discarding several functions), which explore the mutants of a seed query thoroughly. We also prove the correctness of our test oracle to solidify the theoretical foundation of PINOLO.

We implemented our idea as a DBMS testing system and evaluated it using four widely-used and comprehensively tested DBMSs, including MySQL, MariaDB, TiDB, and OceanBase. Compared with the state-of-the-art approaches, PINOLO is more effective in detecting logical bugs. During the 24-hour run, PINOLO can find 41 unique logical bugs, while the three state-of-the-art approaches together can only discover 14 bugs. Upon the submission, 39 out of 41 bugs have been confirmed by developers, showing the great impact of PINOLO on the four real-world DBMSs. In summary, this paper makes the following contributions:

- We introduce the concept of the approximation relation and a series of approximate mutators to resolve the test oracle problem in testing logical bugs in DBMSs.
- We propose a systematic metamorphic testing based approach PINOLO to detecting logical bugs in DBMSs, which leverages the approximate mutators to synthesize approximate queries for a seed query.
- We implement our idea as a DBMS testing system and systematically evaluate it using four widely-used DBMSs. The evaluation results demonstrate the effectiveness of PINOLO in detecting logical bugs.

2 Background

As discussed in § 1, this paper focuses on finding logical bugs in the DBMSs. This section provides several preliminaries as the background, including the concept of the DBMS, the logical bugs in the DBMSs, and the metamorphic testing based approaches for DBMS testing.

2.1 Database Management Systems

The DBMSs are widely used in many modern software systems. They enable the developers to perform various data manipulations, namely insertion, removal, update and search, according to their demands. Here, we concentrate on the relational DBMSs in our work as our target, which are one typical kind of DBMSs. Basically, they are developed on top of the relational model (RM) [7], where data is organized as a collection of tables. Each table is essentially a relation storing the records inserted by the developers, which is a multi-set of tuples from a mathematical perspective. Finally, a database in the DBMS consists of one or more tables, storing the data in a relational manner. In this paper, we use the DBMSs to indicate the relational DBMSs without introducing ambiguity.

There have been an increasing number of DBMSs released by the industry and academia, including MySQL, MariaDB, TiDB, and OceanBase [9, 22, 26, 29]. To interact with DBMSs, SQL [3], which is the most commonly used domain-specific language to store and operate data, was proposed. When retrieving data, developers write SQL queries and send them to DBMSs to get querying results. Each querying result is a multi-set of tuples indicating specific attributes of queried records in the tables. Overall, DBMSs provide an intuitive and flexible way to store and retrieve information, promoting the prosperity of database-backed applications in the real world.

```
--create a table
CREATE TABLE t ( c1 FLOAT );
INSERT INTO t VALUES (-1);

-- queries
(SELECT 1 FROM t WHERE COT(0.2)=0)
UNION ALL (SELECT (BINARY c1 | 0) FROM t);
--result: {0}

(SELECT 1 FROM t WHERE TRUE)
UNION ALL (SELECT (BINARY c1 | 0) FROM t);
--result: {18446744074709551615, 1}

(SELECT 1 FROM t WHERE FALSE)
UNION ALL (SELECT (BINARY c1 | 0) FROM t);
--result: {18446744074709551615}
```

Figure 1: An example of a logical bug in OceanBase.

2.2 Logical Bugs in DBMSs

With the prevalent usage of DBMSs in real-world industrial scenarios, their reliability and correctness have recently been paid increasingly more attention. As complex software systems, DBMSs can have bugs that cause crashes and other unexpected behaviors. Remarkably, the logical bugs are one of the most tricky bugs underlying the DBMSs. When a developer writes a SQL query and executes it upon a database, the returned result may be erroneous, which means that the semantics of the query is not correctly evaluated.

Figure 1 shows a logical bug in OceanBase [25]. We simplify the creation of the table for better demonstration. Specifically, the first query selects the constant value 1 from the table *t* if the cotangent of 0.2 is equal to 0, while the second and the third queries replace the predicates in the where clauses with 1 and 0, respectively. Obviously, the querying result of the first query should be a subset of the one of the second query, and meanwhile, it subsumes the querying result of the third one. However, the actual querying results do not conform to such inclusion relations. As confirmed by the developers of OceanBase, the querying result of the first query is incorrect, which is caused by the simultaneous usage of the set operator **UNION ALL** and the functions **COT** and **BINARY**.

As shown by the above example, logical bugs are more mysterious than system crashes, which have apparent manifestations. In contrast, people are often unaware of logical bugs in DBMSs. Typically, the developers of database-backed applications may realize the abnormal data retrieved from the database, while they are still uncertain whether their application is wrongly implemented or a logical bug of the DBMS is triggered. Therefore, detecting a logical bug in the DBMS has been typically recognized as a problem with both high impact and significant technical challenges.

2.3 Metamorphic Testing

Recent years have witnessed tremendous efforts in resolving the test oracle for logical bug detection in the DBMSs. Notably, the metamorphic testing based approach has been recognized to be state-of-the-art in DBMS testing for logical bug detection [35, 37]. Generally, the metamorphic testing based techniques attempt to construct multiple SQL queries of which the querying results have a specific relation, namely a metamorphic relation. If the querying results violate the metamorphic relation, we can have the confidence that at least one of the queries triggers a logical bug in the tested DBMS. For example, NoREC [34] transforms a query into a form in which the DBMS does not apply optimizations, which yields the test oracle that the two queries should make the tested DBMS return the same result. Besides, TLP [35] gets the equivalent query result by splitting the input query into several sub-queries and merging the results of sub-queries into one. When adapting the metamorphic testing, they take randomly-generated queries as the seed queries and then transform them into other queries to ensure the metamorphic relations, which automates the testing process for logical bug detection.

Unfortunately, the existing effort has not resolved the test oracle problem perfectly. To the best of our knowledge, previous studies only leverage the equivalent transformations, which are supported by the tested DBMS or conducted by their approaches, leaving the functions and operators in the query unchanged. This greatly limits their approaches to exploring the code of the tested DBMSs and thus reduces the chance of finding logical bugs. For example, the logical bug shown in Figure 1 can not be revealed by NoREC [34] and TLP [35], as the transformations preserve all the operators and the functions, still triggering the buggy evaluation process. To improve the capability of the metamorphic testing in finding logical bugs in the DBMSs, we propose a new DBMS testing approach in this work, which relaxes the equivalence relation with a less restrictive metamorphic relation, finally supporting finding more insightful logical bugs.

3 Approximate Query Synthesis

We propose the approximate query synthesis technique PINOLO[1] for detecting logical bugs in the DBMSs. Basically, our insight comes from the intuition that the mutation of specific grammatical constructs can induce the approximation relation between the original query and the mutated one, which can be adopted as the oracle of DBMS testing. Specifically, we start from a randomly generated seed query and mutate several constructs, such as predicates in where clauses, comparison operators, and set operators. Based on the mutation upon any seed query, we can successfully synthesize

[1] Pinolo is the English name of a cartoon character named Pinocchio. Once he tells a lie, his nose will become longer. The relative change in its length is the evidence to verify whether he is lying.

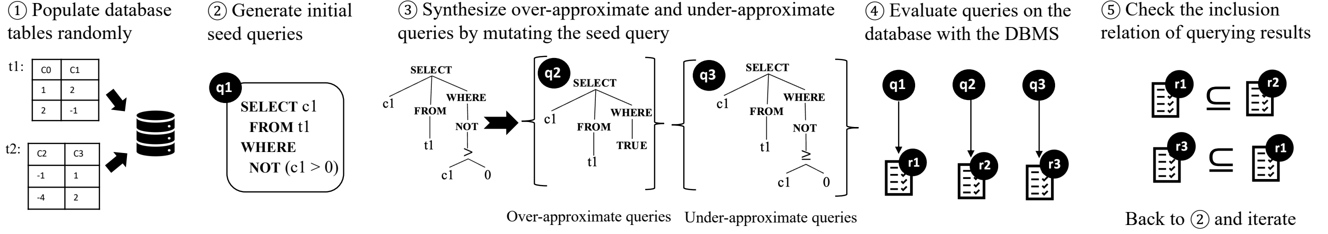


Figure 2: The workflow of PINOLO

a series of queries, of which the return results are the superset or the subset of the result of the seed query. If the seed query and a synthesized query do not produce the results with the expected inclusion relation, we safely conclude the existence of a bug underlying the DBMS. This section presents the system design of PINOLO to show how it resolves the oracle problem in detecting logical bugs in the DBMS and demonstrates the details of mutation-based query synthesis.

3.1 Approach Overview

We demonstrate the overall workflow of PINOLO in Figure 2. In the pre-processing phase, we populate several tables in a database by generating table records randomly, which leverages the existing DBMS random testing technique [27]. After preparing the database, PINOLO first generates a syntactically valid SQL query as the seed query. Then it parses the seed query and traverses its AST to determine whether each grammatical construct can be mutated. The mutations can make PINOLO synthesize several new queries of which the querying results have the inclusion relation with the one of the seed query, achieving the over or under-approximation for the seed query. Based on the synthesized queries and their approximation relation with the seed query, we further evaluate them on the populated database. Any violation of the approximation relation reveals a potential logical bug of the DBMS as at least one of the querying results of the seed query and the synthesized one is incorrect.

The critical component of PINOLO is to automatically generate the pairs of SQL queries with known approximation relations. To show more technical details, we first propose the approximation relation for SQL queries (§ 3.2), and then demonstrate how to synthesize the queries with approximation relations based on mutations (§ 3.3 and § 3.4). We summarize our design and highlight the advantage of PINOLO (§ 3.5).

3.2 SQL Query Approximation

In this work, we concentrate on the syntax of SQL queries shown in Figure 3. Basically, a SQL query can be a select-from-where query or the result of the set operation upon subqueries. The logical connectives and arithmetic operators make the query support depicting sophisticated predicates and

$$
\begin{aligned}
\textit{Relation } R &:= t \mid q \mid r_1 \otimes r_2 \mid r_1 \textbf{ JOIN } r_2 \\
\textit{Query } Q &:= q_1 \otimes q_2 \mid \textbf{SELECT } a \textbf{ FROM } r\ (\textbf{WHERE } p)^? \mid \\
&\quad\ \textbf{SELECT DISTINCT } a \textbf{ FROM } r\ (\textbf{WHERE } p)^? \\
\textit{Pred } P &:= \ell_b \mid e_c \mid p \textbf{ IS } \ell_b \mid p \textbf{ IS NOT } \ell_b \\
&\quad \mid p_1 \textbf{ AND } p_2 \mid p_1 \textbf{ OR } p_2 \mid \textbf{NOT } p \\
\textit{CExpr } E_c &:= e_{a1} \odot e_{a2} \mid e_a \odot \textbf{ALL}(r) \mid e_a \odot \textbf{ANY}(r) \\
\textit{AExpr } E_a &:= \ell_n \mid c \mid e_{a1} \oplus e_{a2} \mid f(e_a) \\
\textit{SOp } \otimes &:= \textbf{UNION} \mid \textbf{UNION ALL} \mid \textbf{INTERSECT} \mid \textbf{MINUS} \\
\textit{COp } \odot &:= \ > \ \mid \ < \ \mid \ \geq \ \mid \ \leq \ \mid \ == \ \mid \ \neq \\
\textit{AOp } \oplus &:= \ + \mid - \mid \times \mid \div \mid \cdots \\
\textit{BLit } L_b &:= \textbf{TRUE} \mid \textbf{FALSE} \qquad \textit{NLit } L_n := \ell_n \\
\textit{Attrs } A &:= c \mid e_a \mid e_a\ a \\
\textit{Table } T &:= t \qquad \textit{Attr } C := c
\end{aligned}
$$

Figure 3: The syntax of SQL queries

quantities. Without the loss of generality, we only instantiate an arithmetic expression with an integer literal, an attribute, the result of an arithmetic operator, and the result of a SQL built-in function. Notably, we can also utilize the keywords **ALL** and **ANY** to support advanced comparison between an arithmetic expression and the numeric attributes.

To resolve the test oracle problem in the DBMS testing, we follow the spirit of metamorphic testing and propose the approximation relations among the SQL queries. In what follows, we first formulate the concept of the approximation relation and then characterize the form of the SQL queries that can have approximation relations with other queries.

Definition 3.1. (Approximation Relation) Given a database D, a SQL query q_1 is the over-approximation of q_2 over D, denoted by $q_2 \trianglelefteq_D q_1$, if and only if $R(q_2, D) \subseteq R(q_1, D)$. Here, $R(q, D)$ is the return result set of q upon the database D, which is essentially a multi-set. $\subseteq$ is the inclusion relation between two multi-sets. We say q_2 is the under-approximation of q_1 over D, denoted by $q_1 \trianglerighteq_D q_2$, if and only if $q_2 \trianglelefteq_D q_1$.

Intuitively, the approximation relation between two SQL queries indicates the inclusion relation of their querying results. If we construct a pair of SQL queries (q_1, q_2) such that

$q_1 \trianglelefteq_D q_2$ or $q_1 \trianglerighteq_D q_2$, we can utilize the approximation relation as an instantiation of the metamorphic relation, which serves as the test oracle for DBMS testing.

Example 3.1. Assume that we have a database $D = \{\text{t1}\}$, where the schema of t1 is (c1) and $\text{t1} = \{(-1),(0),(1)\}$. Consider the following three queries.

$$q_1: \textbf{SELECT}\ \text{c1}\ \textbf{FROM}\ \text{t1}\ \textbf{WHERE NOT}\ (\text{c1} > 0)$$
$$q_2: \textbf{SELECT}\ c1\ \textbf{FROM}\ \text{t1}\ \textbf{WHERE TRUE}$$
$$q_3: \textbf{SELECT}\ \text{c1}\ \textbf{FROM}\ \text{t1}\ \textbf{WHERE NOT}\ (\text{c1} \geq 0)$$

The first query q_1 selects all the non-positive values of the attribute c1 of the table named t1. The second query q_2 selects all the values of the attribute c1. The third query q_3 selects all the values of the attribute c1 that are not large than or equal to 0. Obviously, their querying results, denoted by $R(q_1,D)$, $R(q_2,D)$, and $R(q_3,D)$, respectively, have the relation that $R(q_3,D) \subseteq R(q_1,D) \subseteq R(q_2,D)$, implying $q_3 \trianglelefteq_D q_1 \trianglelefteq_D q_2$, i.e., $q_2 \trianglerighteq_D q_1 \trianglerighteq_D q_3$.

To sum up, the syntax shown in Figure 3 characterizes the search space of constructing a pair of queries with an approximation relation. Given a seed query in the syntax, we can always obtain a query upon a smaller/larger relation or with a stronger/weaker where clause, which induces a subset/superset of the return result of the seed query, achieving the under/over-approximation of the given seed query. Therefore, it is feasible to automatically generate the queries that have the approximation relation with a specific query q by mutating the query q, which trims/enlarges the relation or strengthens/weakens the predicate in the seed query. In this way, we can resolve the test oracle problem by synthesizing queries with approximation relations.

3.3 Approximate Mutators

Based on the key insight in § 3.2, we propose to resolve the test oracle problem by constructing SQL queries with the approximation relation. Specifically, we can always obtain the approximation relation if we transform a query to another one preserving the set inclusion relation of the relations and the implication relation of the predicates. According to high-level intuition, we propose the concept of the approximate mutator as follows, which is the fundamental ingredient of the approximate query synthesis in § 3.4.

Definition 3.2. (Approximate Mutator) An approximate mutator is a mapping from a SQL query q_1 to a query q_2 such that $q_1 \trianglelefteq_D q_2$ or $q_1 \trianglerighteq_D q_2$.

Essentially, an approximate mutator transforms a SQL query into another such that they have the over or under-approximation relation. We notice that a relation can be derived from other relations, e.g., the results of a select-from-where query and set operations, while compound and atomic logical expressions pose restrictions over relations. Therefore, we propose three categories of approximate mutators, which are shown in Table 1.

Table 1: Some representative approximate mutators. Transforming the construct **C1** into **C2** achieves the under-approximation, while transforming the construct **C2** into **C1** achieves the over-approximation.

Type	C1	C2
Relation	**SELECT** a **FROM** r	**SELECT DISTINCT** a **FROM** r
	r_1 **UNION ALL** r_2	r_1 **UNION** r_2
	r_1 **UNION** r_2	r_1
	r_1 **UNION** r_2	r_1 **INTERSECT** r_2
	r_1	r_1 **MINUS** r_2
Predicate	p	**FALSE**
	p **IS** ℓ_b	**FALSE**
	p **IS NOT** ℓ_b	**FALSE**
	TRUE	p
	TRUE	p **IS** ℓ_b
	TRUE	p **IS NOT** ℓ_b
Comparison expression	$e_{a1} \leq e_{a2}$	$e_{a1} = e_{a2}$
	$e_{a1} \geq e_{a2}$	$e_{a1} = e_{a2}$
	$e_{a1} \leq e_{a2}$	$e_{a1} < e_{a2}$
	$e_{a1} \geq e_{a2}$	$e_{a1} > e_{a2}$
	$e_{a1} \neq e_{a2}$	$e_{a1} < e_{a2}$
	$e_{a1} \neq e_{a2}$	$e_{a1} > e_{a2}$
	$e \odot$ **ANY**(r)	$e \odot$ **ALL**(r)

Concretely, the mutators alter the relations and predicates in a SQL query, and meanwhile, mutate the comparison expressions, which are often atomic constraints in a predicate, achieving the approximation relation between the queries before and after the mutation. For each row, if we replace the SQL grammatical construct in the second column with the one in the third column, we can obtain a query that under-approximates the original one; if we replace the one in the third column with the one in the second column, we can obtain a new query that over-approximates the original query. Now we provide more explanations on the approximate mutators.

- **Mutating relations.** Using **DISTINCT** in a select-from-where query removes the duplicate values in the querying result, which under-approximates the original query. The set operator **UNION ALL** preserves the duplicate values and the operator **UNION** does not, so replacing the former with the latter ensures the under-approximation relation between the new query and the original one. Other mutators altering relations are fairly simple.
- **Mutating predicates.** For an arbitrary predicate p, we can strengthen it by mutating it to **FALSE** and weaken it by mutating it to **TRUE**. The logical implication would pose more or less restrictive constrain upon the tuples in the relations, which finally achieve the under-approximation or over-approximation, respectively.
- **Mutating comparison expressions.** For each comparison expression as an atomic constraint, we can alter its comparison operator to strengthen or weaken the constraint induced by the expression. For example, replacing

$\geq$ with $=$ makes the new expression more restrictive than the original one. Also, mutating the keyword **ANY** with **ALL** also induces a stronger predicate in the query.

Example 3.2. For the simpler SQL query q_1 in Example 3.1, we can mutate it by replacing the negation with **TRUE** and altering $>$ to $\geq$, which yield two queries q_2 and q_3 that over-approximate q_1, respectively. We can further consider a more complex SQL query as follows.

$\widetilde{q}$: **SELECT** 1 **FROM** t1 **WHERE**
(**NOT** (**FROM_DAYS**(1) = **ALL**(**SELECT** c1 **FROM** t1)))

We can mutate the predicate in the where clause of $\widetilde{q}$ to **TRUE** to over-approximate the query $\widetilde{q}$. Also, mutating **ALL** to **ANY** weakens the comparison expression in the negation, and thus, strengthens the predicate in the where clause, yielding an under-approximation of the query $\widetilde{q}$.

Notably, the approximate mutator proposed in this section is a general concept. The mutators shown in Table 1 are just several instances of the mutators, while we can further define more mutators to enable us to obtain the queries with approximation relations more flexibly. Actually, we provide a systematic framework to instantiate such mutators in these categories. The complete list of the instantiated mutators, including REGEXP and IN operators, has been published online [31].

3.4 Mutation-based Query Synthesis

Leveraging the approximate mutators in § 3.3, we can finally propose the approximate query synthesis algorithm by applying the mutators upon a seed query. This section demonstrates the technical details of the synthesis algorithm on how to generate two sets of SQL queries that over-approximate and under-approximate a seed query, respectively. We also formulate our test oracle with a theorem as the theoretical guarantee for the approximation relations among the seed query and the synthesized ones.

Algorithm 1 shows the mutation-based query synthesis algorithm. Initially, it takes a seed query as the input, parses the query, and generates an AST of the query to facilitate further mutations (Line 2). Basically, it traverses the AST in a top-down manner, during which it identifies the potential SQL constructs for the mutation (Line 3–Line 4). Consider synthesizing the queries that under-approximate the seed query as an example, where *kind* is set to be *UNDER* (Line 4). Specifically, it processes each SQL construct in two ways.

- When encountering the SQL construct (r_1 **MINUS** r_2), it attempts to trim the relation r_1 and enlarge the relation r_2 so that the difference of the two relations can be trimmed (Line 12–Line 15). Similarly, it strengthens the predicate p for (**NOT** p) and (p **IS NOT TRUE**), and also enlarges the relation r for the comparison expression $e \odot$ **ALL**(r) (Line 16–Line 24).

Algorithm 1: Mutation-based query synthesis

```
1  Procedure synthesizeApproximateQueries(q):
2      τ ← parseQuery(q) ;
3      Q_over ← mutate(q, τ, OVER) ;
4      Q_under ← mutate(q, τ, UNDER) ;
5      return (q, Q_over, Q_under);
6  Procedure mutate(u, τ, kind):
7      if kind == OVER then
8          negKind ← UNDER;
9      else
10         negKind ← OVER;
11     S ← applyApproxMutator(q, τ, kind) ∪ {q};
12     if u : (r1 MINUS r2) then
13         S'1 ← mutate(r1, getAST(r1), kind);
14         S'2 ← mutate(r2, getAST(r2), negKind);
15         S ← S ∪ {r'1 MINUS r'2 | r'1 ∈ S'1, r'2 ∈ S'2} ;
16     else if u : (NOT p) then
17         S' ← mutate(p, getAST(p), negKind);
18         S ← S ∪ {NOT p' | p' ∈ S'};
19     else if u : (p IS NOT TRUE) then
20         S' ← mutate(p, getAST(p), negKind);
21         S ← S ∪ {p' IS NOT TRUE | p' ∈ S'};
22     else if u : e ⊙ ALL(r) then
23         S' ← mutate(r, getAST(r), negKind);
24         S ← S ∪ {e ⊙ ALL(r') | r' ∈ S'} ;
25     else
26         Γ ← getSubASTs(u) ;
27         Π ← ⊥;
28         foreach (v, τ_v) in Γ
29             Π[(v, τ_v)] ← mutate(v, τ_v, kind);
30         S ← S ∪ concat(q, Γ, Π) ;
31     return S;
```

- For other SQL constructs, it trims each relation and strengthens each predicate and comparison expression appearing in the constructs. Lastly, it composes each mutated constructs together by the function *concat* to obtain the ASTs of the synthesized queries under-approximating the seed query (Line 26–Line 30).

By applying the approximate mutators during the AST traversal, Algorithm 1 finally synthesizes two sets of queries on the fly, which are syntactically valid and have the approximation relation with the seed query q.

Example 3.3. Consider the query $\widetilde{q}$ in Example 3.2 as the seed query. We show how to synthesize the queries that under-approximate $\widetilde{q}$. After generating its AST, which is shown by the leftmost tree in Figure 4, Algorithm 1 examines each SQL construct in a top-down manner. When encountering the predicate in the where clause, we can mutate the predicate, which is a logical negation, to **FALSE**, or strengthen the predicate in the logical negation. For the latter case, we can further mutate the comparison expression in the negation to **TRUE**, mutate the comparison operator with $\geq$, or replace **ALL** with **ANY**, which finally weakens the logical negation. Finally, we can

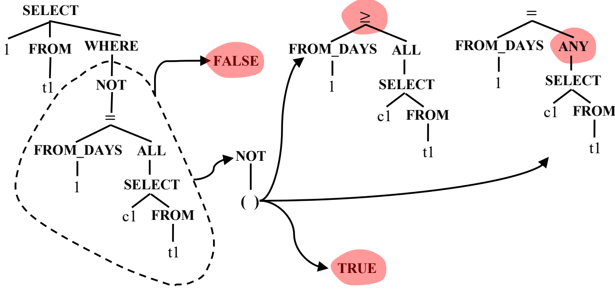


Figure 4: An example of synthesizing queries under-approximating the query $\widetilde{q}$

obtain four queries that under-approximate $\widehat{q}$. Particularly, one of the synthesized queries is as follows:

$$\widetilde{q}' : \textbf{SELECT } 1 \textbf{ FROM } \text{t1} \textbf{ WHERE } (\textbf{NOT } (\textbf{FROM_DAYS}(1) \geq \textbf{ALL}(\textbf{SELECT } \text{c1} \textbf{ FROM } \text{t1})))$$

Fortunately, we find that $R(\widetilde{q}', D)$ is not subsumed by $R(\widetilde{q}, D)$, indicating that the querying result of $\widetilde{q}$ or $\widetilde{q}'$ is incorrect, which is confirmed by the developers of MySQL [21].

It is worth mentioning that we have to restrict the values in the database tables not to be null. Any comparison between a non-null value and a null value can introduce an unknown value of the comparison expression, denoted by **NULL** without the ambiguity, which is smaller than **TRUE** but larger than **FALSE** in the logical order. However, we can notice that the predicate (p **IS NULL**) evaluates to **FALSE**, **TRUE**, or **FALSE**, if p is **TRUE**, **NULL**, or **FALSE**, respectively, which indicates that strengthening or weakening the predicate p does not always strengthen or weaken the predicate p **IS NULL**. In this case, we cannot ensure the expected approximation relation between the seed query and the synthesized queries in Q_{over} and Q_{under}. Formally, we state the following theorem to formulate our test oracle.

Theorem 3.1. (Test Oracle) Assume that the database D does not contain any table storing null values. Taking the query q as a seed query, Algorithm 1 can always synthesize two sets of queries Q_{over} and Q_{under} such that:

- For any $q' \in Q_{over}$, we have $q \trianglelefteq_D q'$, i.e., $q' \trianglerighteq_D q$.
- For any $q' \in Q_{under}$, we have $q' \trianglelefteq_D q$, i.e., $q \trianglerighteq_D q'$.

To solidify the theoretical foundation of our test oracle, we sketch the proof of the theorem briefly. First, the fact that the database tables do not contain null values implies that the evaluation results of any expressions, including comparison expressions and predicates, are not evaluated to unknown values, and the intermediate relations, such as the results of the join and the union operator, do not contain null values. Second, it is trivial to prove that the approximate mutators applied to different constructs finally yield a weaker predicate or larger relation when *kind* is *OVER* in the absence of null values. A similar argument also holds when *kind* is *UNDER*. Therefore, we can prove the approximation relation between each synthesized query and the seed query based on the principle of structural induction.

3.5 Summary

PINOLO automates the DBMS testing via the approximate query synthesis, which discovers underlying logical bugs in the DBMSs. The syntax in Figure 3 ensures the syntactical validity of the seed queries, and furthermore, guarantees that the synthesized queries have valid SQL syntax. Meanwhile, our approximate mutators enable us to obtain the approximation relation between the seed query and the synthesized ones, which perfectly resolves the test oracle problem. Compared with the existing techniques [34–36], PINOLO considers more SQL features, such as set operators, arithmetic expressions, sub-queries, etc. The expressive syntax permits us to test the DBMS more thoroughly and discover more logical bugs reported in previous studies, which will be evidenced by our experiments. Besides, the approximate operators can aggressively mutate the seed query, which may remove the buggy operators and functions, revealing the logical bugs more thoroughly than existing techniques. Lastly, it is worth noting that PINOLO provides a general framework for discovering the logical bugs in the DBMS. We can further extend the syntax of SQL queries and instantiate more approximate mutators, which can promote the capability of discovering the logical bugs triggered by sophisticated SQL queries.

4 Implementation

We implemented our approach PINOLO as a DBMS testing system, which was written in GO with 8,055 lines of code. The source code of our tool is hosted in the github repository [2]. Next, we present more details of our implementation decisions that are important for the outcome of our experiments.

Database population. We randomly generate the database tables by leveraging an existing tool, GO-RANDGEN [27]. Following the best practice summarized in the prior study [36], we restrict the number of table records to be no more than 30. Besides, we randomly generate the tables with the same attributes, which makes the join operator yield a non-trivial result. Particularly, we avoid null values in any table, as the test oracle requires non-null values as a prerequisite, which is stated in Theorem 3.1.

Seed query generation and parsing. To generate seed queries as the input of our synthesis algorithm, we utilize GO-RANDGEN [27] to automatically produce seed SQL queries. Specifically, we use the general-purpose parser generator BISON [13] to write a context-free grammar file describing the SQL syntax with a series of production rules. We provide

[2] https://github.com/qaqcatz/impomysql.git

this grammar file to GO-RANDGEN, so that it can generate the queries by searching each production rule randomly. It then heuristically selects the terminal or non-terminal symbols to avoid exceeding the limitation of recursion. Moreover, we permit users to write embedded LUA code blocks in the grammar file to further restrict the form of seed queries, which can ensure the successful query execution, e.g., the number of columns in the two queries of **UNION** should be equal. To apply the approximate mutators to seed queries, we utilize another tool PINGCAP PARSER [28], which accepts the same context-free grammar as the one for the seed query generation, to generate ASTs of seed queries for the mutation.

Approximate mutator instantiation. As mentioned at the end of § 3.3, we instantiate a series of approximate mutators for the relations, predicates, and comparison expressions in a given query. Each approximate mutator consists of two SQL grammatical constructs, which indicate the construct after the over and under-approximation, respectively. To cover most features of DBMSs, we instantiate 25 approximate mutators in total, including the approximate mutators demonstrated in Table 1. Among them, 5, 6, and 14 approximate mutators correspond to the mutations of the relations, predicates, and comparison expressions, respectively. Apart from the approxmiate mutator in Table 1, we also include 7 mutators supporting **LIKE**, **REGEXP**, **IN**, and **BETWEEN**.

Bug report post-processing. After synthesizing queries, PINOLO obtains the querying results by evaluating queries on the populated database in the tested DBMSs. It is noted that inconsistent querying results frequently occur in our testing process. For example, during a 24-hour testing period, 46,772 inconsistent query pairs are generated for MySQL. The large number of such query pairs makes the process of confirming and fixing bugs quite verbose. To make the testing results easier to understand, we borrow the idea of *delta debugging* [56] to localize the root cause of the inconsistent returned results of each query pair. Specifically, we associate each problematic query pair with a release version of the under-test DBMS's code base, the earliest one where inconsistent query results appear. We denote such release version with respect to a given query pair as the bug-inducing version. Further, two bugs are considered the same if their bug-inducing versions are the same. Based on our interactions with DBMS developers, our bug reports and the release version's change lists can help developers pinpoint culprit updates easily.

5 Evaluation

To evaluate the performance of our approach PINOLO in detecting logical bugs in the popular real-world DBMSs, we design the following research questions:

- **RQ1:** How many logical bugs in real-world DBMSs can be detected by PINOLO?
- **RQ2:** Can PINOLO outperform the state-of-the-art logical bug detection techniques?
- **RQ3:** How does the randomness introduced by the seed query generation affect the performance of PINOLO?

5.1 Experiment Setup

Environment. We conducted the experiments on one server with 104-cores Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz and 500 GB memory. The server runs Ubuntu 18.04 system that uses Linux kernel version: 5.4.0-135-generic. To ensure fair comparisons, we allocated four threads for each DBMS testing in our following experiments.

Tested DBMS. Our focus was on testing four widely-used and large-scale open-source DBMSs: MySQL, MariaDB, TiDB, and OceanBase. There are two main reasons for the subject selection. First, these selected DBMSs are representative DBMSs from phenomenal open-source and/or commercial products: MySQL and MariaDB are the two most well-known open-source DBMSs; OceanBase is a mature commercial database product from Ant Group; TiDB is developed by PingCap Inc. They are also commonly used in the evaluation of previous studies [34–36]. Moreover, we admit and discuss potential limitations introduced by our selection of DBMS systems in § 6. Second, we chose a DBMS whose SQL syntax is compatible with MySQL as an evaluation subject to reduce the implementation effort. This is because, although our approach can be generalized to other DBMSs, the implementation of the seed query generation and parsing (See § 4) requires a grammar file that describes the SQL syntax of a tested DBMS. To obtain timely feedback from developers, we tested the latest release versions of the selected DBMSs: MySQL 8.0.31, MariaDB 10.11.1, TiDB 6.4.0, and OceanBase 4.0.0.

Baseline. We compared PINOLO with the three state-of-the-art logical bug detection techniques, namely PQS [36], NOREC [34], and TLP [35], respectively, which correspond to three kinds of test oracles. Similar to our approach, these baselines also require knowledge about the SQL syntax of different DBMSs for seed query generation and parsing. Unfortunately, their implementations cannot support all the selected DBMSs. We also sought help from their authors, but they still could not fix the problems before the paper submission. Therefore, we skipped the evaluation of the baselines on the unsupported DBMSs. Moreover, we tried to use the same random seed as the baselines. However, we found that they can neither export their random seeds nor import the random seeds provided by users. Therefore, we generated the random seeds by ourselves for PINOLO. To understand the impacts of the random seed query generation, we investigated how PINOLO is robust to such randomness in § 5.4.

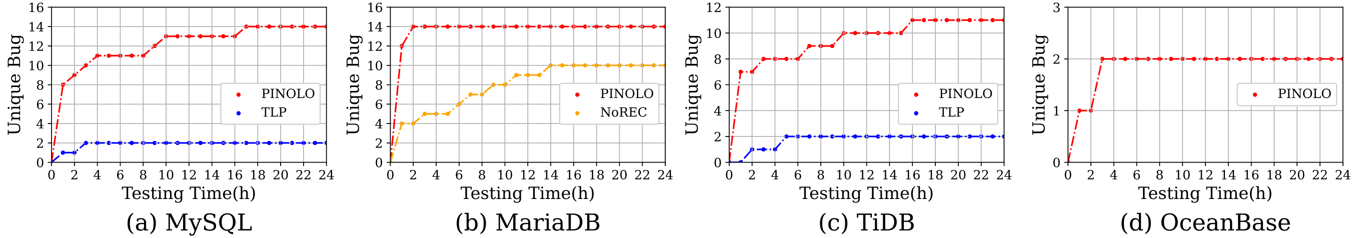


Figure 5: Comparison between PINOLO and the baselines. (a)-(d) show the number of unique logical bugs over time (24 hours) on each DBMS. We ignore the results of the baselines on the unsupported DBMSs and the baselines finding no bugs.

Table 2: The demographics of the DBMSs under the test

DBMS	Version	GitHub Stars	LOC	First Release
MySQL	8.0.31	8.6K	4,766,086	1995
MariaDB	10.11.1	4.6K	3,727,410	2009
TiDB	6.4.0	33.1K	985,518	2017
OceanBase	4.0.0	5.1K	2,722,881	2021

Table 3: Applicability of existing logical detection techniques and PINOLO for the selected DBMSs

DBMS	PQS	NOREC	TLP	PINOLO
MySQL	✓	×	✓	✓
MariaDB	×	✓	×	✓
TiDB	×	×	✓	✓
OceanBase	✓	✓	✓	✓

5.2 Effectiveness of PINOLO

We used PINOLO to test the latest version of MySQL, MariaDB, TiDB, and OceanBase for 24 hours. Table 4 summarizes the results of PINOLO. The column **All** shows the number of problematic query pairs that induce unexpected results, which indicate the existence of logical bugs. As we can see, PINOLO discovered a large number of problematic query pairs, ranging from 4,675 to 46,772 for the tested DBMSs. However, we found that most of these pairs can be attributed to the same bug. To relieve the developers from the heavy burden of checking the duplicate bugs, we leverage the bug-inducing version to deduplicate the bugs (See § 4). The column **Unique** shows the number of bug reports after deduplication, which is significantly smaller than the value in column **All**, ranging from 2 to 14. We submitted the deduplicated bugs to the developers for confirmation. The column **Verified** shows the number of bug reports that have been verified by developers, ranging from 2 to 14.

Table 4: The number of logical bugs found by PINOLO.

DBMS	All	Unique	Verified
MySQL	46,772	14	14
MariaDB	42,208	14	12
TiDB	5,268	11	11
OceanBase	4,675	2	2

In total, PINOLO found 41 unique logical bugs in these DBMSs, 39 of which have been confirmed by developers. For MySQL, TiDB, and OceanBase, all of the detected bugs have been confirmed by developers. For MariaDB, twelve out of fourteen bugs have been confirmed, while the rest are still waiting for the investigation. We sampled several bug reports of MariaDB submitted by others and found that developers typically take a much longer time to handle the bugs. To keep track of the status of our reported bugs, we release the bug list in a public GitHub repository[3].

Answer to RQ1: PINOLO discovers 41 unique bugs on MySQL, MariaDB, TiDB, and OceanBase, 39 of which have been confirmed by DBMS developers.

5.3 Comparisons on Detecting Logical Bugs

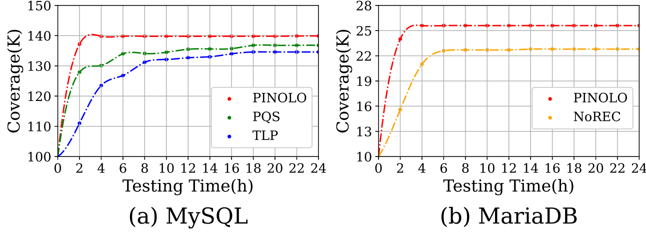
Logical bug detection. We compared PINOLO with the three state-of-the-art baselines, i.e., PQS, NOREC, and TLP. We ran all methods with a time budget of 24 hours. The comparison results are shown in Figure 5. Note that the baselines do not support all the DBMSs, and thus we only concentrated on the comparison between PINOLO and the runnable baselines with respect to each DBMS.

For MySQL, PINOLO detected 14 logical bugs, while TLP only discovered 2 bugs. For MariaDB, PINOLO detected 14 bugs, while NOREC discovered 10 bugs. For TiDB, PINOLO detected 11 bugs, while TLP only discovered 2 bugs. For OceanBase, PINOLO can detect 2 bugs, while the baselines cannot find any. We also manually verified the overlap in the bugs detected by PINOLO and the baselines. For MariaDB, 4 out of 10 bugs detected by NOREC can also be found by PINOLO. For TiDB, 1 out of 2 bugs detected by TLP can also be found by PINOLO. There are no bugs detected by PQS.

Figure 5 shows the logical bug detection progress over time for PINOLO and the baselines. We found that PINOLO is more efficient in finding logical bugs compared to all of the baselines. Within one hour, PINOLO was able to detect 57.1% (8/14) of bugs on MySQL, 85.7% (12/14) on MariaDB, 63.6% (7/11) on TiDB and 50% (1/2) on OceanBase.

Code coverage. To understand why PINOLO can find more

[3]https://github.com/qaqcatz/impomysql_bugreports



(a) MySQL (b) MariaDB

Figure 6: Code coverage comparison between PINOLO and the three baselines over 24 hours

logical bugs than all other methods, we used GCOV [30] to compute the line coverage achieved by PINOLO and all others. Note that we are unable to provide code coverage results for TiDB and OceanBase. As TiDB is developed in GO, we cannot find any feasible tool to support the program instrumentation or the code coverage profile for system test [4]. For OceanBase, the instrumented binary has to be deployed by the specific tool OBDEPLOY [24], which breaks the functionality of code coverage profiling.

Figure 6 shows the code coverage over time for MySQL and MariaDB. The results show that PINOLO achieves a higher line coverage than the three baselines. For MySQL, the improvement of PINOLO over PQS and TLP is 2.2% (3,008 lines) and 4.0% (5,316 lines), respectively. For MariaDB, the improvement of PINOLO over NOREC is 12.4% (2,835 lines).

Code coverage is well recognized as an approximation of testing capability, because a bug cannot be detected if its buggy code is not executed. However, larger code coverage does not mean more bugs. To better understand the impact of larger code coverage achieved by PINOLO, we further investigated whether there were some bugs whose buggy code is uniquely covered by PINOLO. TiDB#40015 [42] is a typical example. The root cause of this bug is the improper exception handling in the function *vecGetDateFromString*, which has been covered by PINOLO but missed by other baselines during the 24-hour running.

Answer to RQ2: Compared with the state-of-the-art techniques, PINOLO can find more unique logical bugs and achieve higher line coverage.

5.4 Impacts of The Seed Query Generation

PINOLO uses GO-RANDGEN, which takes a random seed to generate a set of seed queries. To understand whether the randomness affects the efficiency and effectiveness of PINOLO, we conducted the experiments which used GO-RANDGEN with five different random seeds to generate five sets of seed queries. We then ran PINOLO under each set and compared their performance on detecting logical bugs.

[4]Go 1.20 plans to cover the features of program instrumentation and code coverage profile, but will be available after Feb 2023 [15].

Table 5: The numbers of the discovered logical bugs when feeding different seed queries to PINOLO

DBMS	Seed1	Seed2	Seed3	Seed4	Seed5	Common
MySQL	14	18	16	16	17	11
MariaDB	14	16	13	13	13	10
TiDB	11	9	11	11	13	9
OceanBase	2	2	2	2	2	2

Table 6: Importance of the logical bugs found by PINOLO

DBMS	Severity		Bug impact duration		
	S2	S3	<1 year	1∼5 years	5∼10 years
MySQL	6	8	1	7	6
MariaDB	9	3	1	7	4
TiDB	8	3	5	6	0
OceanBase	-	-	1	1	0

Figure 7 shows the progress of detecting unique logical bugs over time for the five sets of seed queries. We found that the growth trend of the number of unique bugs over time is similar under different sets. Table 5 shows more details about these detected logical bugs. Among the five sets of random seed queries, the common bugs, of which the numbers are shown in the column **Common**, account for an average of 68.4%, 72.9%, 82.9%, and 100% of the total bugs on MySQL, MariaDB, TiDB, and OceanBase respectively. This result shows that PINOLO can find different unique logical bugs via different sets of random seed queries.

Answer to RQ3: The randomness introduced by the seed query generation does not have a significant impact on the overall bug detection performance of PINOLO. Meanwhile, the different random seed queries can benefit PINOLO in detecting different bugs.

5.5 Discussion

Bug Importance. To understand the importance of the bugs found by PINOLO, we investigated their severity and the impact duration. The results are shown in Table 6. The column **Severity** indicates the severity of the bugs labeled by developers. Our reported bugs were classified as the levels of S2 and S3, which represent the second and third highest severity levels, respectively. Typically, the S2 level indicates a severe loss of service or missing significant functionality, while the S3 level indicates a minor loss of service or inconvenient usage. Note that there is no severity level in the bug tracking system of OceanBase, so we skipped the discussion for OceanBase. We discovered 6, 9, and 8 bugs in the S2 level in MySQL, MariaDB, and TiDB, respectively, which account for 62.2% of the bugs in the three DBMSs. The bugs in the S3 level are less severe, but developers still considered them to be fixed

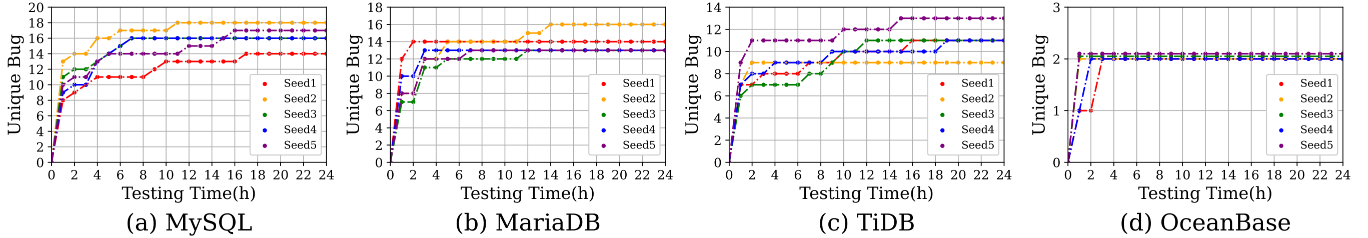


Figure 7: The number of unique logical bugs over time for each group of the random seed queries on the tested DBMSs

necessarily. For example, we received an appreciation from the MySQL developers in one of the bug reports with the S3 level: "*Thank you for your contribution. It is our standpoint that all bugs should be fixed, whether major or minor.*"

Table 6 also shows the **bug impact duration**, which is computed as the interval between the time of the bug-inducing version and the bug reporting time. Surprisingly, we found that 10 (25.6%) and 21 (53.8%) of bugs have lasted for 5-10 and 1-5 years, respectively. Specifically, the two earliest bugs of MySQL [20] and MariaDB [17] can be traced back to 2014. This result indicates that the logical bugs are typically difficult and slow to be found, which is also consistent with the findings of the prior study [35].

False Positive and False Negative. According to Theorem 3.1, PINOLO will not produce false positives in theory. However, we observe two bugs that have not been confirmed by developers for more than five months, and thus suspect that they are false positives. We manually inspected the two cases. In one bug report, a query returns "0", while the approximated query returns "-0". In the other one, a query returns "0", while the approximated query returns "-0.001". Although PINOLO considers the above inconsistency results as bugs, developers may have a higher tolerance for such inconsistencies. These reports allowed us to refine our implementation to reduce false positives in the future.

In terms of false negatives, we observed 9 cases that are detected by the baseline approaches but cannot be detected by PINOLO. This is mainly due to the DBMS features that are currently not supported by PINOLO. Specifically, there are six, two and one bugs that are related to aggregate functions, left/right join, and three-valued logic, respectively.

Limitations and Future Work. We currently do not support all features of DBMSs, such as aggregate functions, window functions, and left/right join. This is because, these features may break the approximation relation (Definition 3.1). For example, aggregate functions typically map a set of values into a single value (e.g., sum, average, maximum, minimum, and so on), and the mapped value will not preserve the inclusion relation of the original sets, thus breaking the approximation relation. In our future work, we will design new approximation relations to support more features. In addition, we intend to explore the application of metamorphic testing in other system software domains (such as networking and distributed systems [1, 32, 43, 44]).

As shown in § 5.4, different random seed queries can benefit PINOLO in detecting different bugs. This indicates that adjusting seeds dynamically is helpful to make PINOLO find more bugs. Therefore, in the future, we will consider to integrate PINOLO into the fuzzing framework to better prioritize the seed selection and enhance the bug detection capability.

Another possible direction for the future exploration is bug deduplication. In this work, we determine whether two bugs are duplicated by checking their bug-inducing versions. However, the bug-inducing version is an approximation of root causes, which would misclassify the bugs. This is because a release version of DBMS would introduce numerous bugs, which lead to multiple problematic query pairs. In the future, we will consider leveraging the spectrum-based fault localization techniques or mutation testing to improve the precision of discovering the root cause of the bugs, so as to improve the precision of bug deduplication.

6 Threats to Validity

The threat to internal validity is primarily associated with the implementation of our approach. To mitigate this concern, we have employed several DBMSs to cross-check whether the mutants generated by PINOLO accurately represented over-approximations or under-approximations of the seed query.

The threat to external validity lies in the representative of the evaluation subjects. Our proposed approach was evaluated on a restricted set of DBMSs, as explained in the evaluation section. As a result, the conclusion drawn in this paper may be limited. However, we believe that it is non-trivial to detect new bugs in these selected DBMSs, as they have been thoroughly tested by SQLancer [33]. In the future, we will extend the implementation of PINOLO to support the evaluation of additional open-source and commercial DBMSs.

7 Related Work

PINOLO is an unique DBMS testing system for finding logical bugs in DBMSs, but draws inspiration from several areas in the literature, including DBMS testing, metamorphic testing, differential testing, and grammar fuzzing.

DBMS testing. Recent efforts on DBMS testing focus on various aspects of DBMSs. Most of them target discovering logical bugs in the relational DBMS [34–36]. They use specific metamorphic relations or multiple implementations as the oracles and generate syntactically valid SQL queries for metamorphic testing [5] or differential testing [18]. Some also attempt to improve the coverage of the DBMSs and leverage fuzzing techniques to enumerate the queries in various forms [10, 46, 49, 57]. PINOLO uses a new design, termed approximate query synthesis, and does not use other DBMS implementations as the oracle.

Metamorphic testing. Metamorphic testing [5] has become more and more popular over the past decade. It has been used in testing many software systems, such as compilers [16, 50], SMT solvers [48, 54], DBMSs [35, 36], and AI systems [2, 47, 55]. Metamorphic testing uses one type of metamorphic relations to compare outputs produced by a seed input and a mutated one. As long as the two outputs violate the specific metamorphic relation, then at least one of the two inputs yields a wrong result [4]. PINOLO utilizes an instantiation of a metamorphic relation, i.e., the approximation relation, as an effective testing oracle for discovering logical bugs in the DBMSs. Similar to the skeleton approximation enumeration in the SMT solver testing [54], PINOLO performs the over-/under-approximation of the seed queries. However, PINOLO has to deal with more sophisticated syntax and supports more flexible mutations so that more buggy operators and functions can be removed. Therefore, it is able to detect insightful logical bugs that existing approaches, such as NOREC [34] and TLP [35], fail to discover.

Differential testing. Apart from metamorphic testing, differential testing provides another testing paradigm for resolving the oracle problem in software testing [6, 40, 52, 53]. Utilizing another software system with the same functionality as an oracle implementation, differential testing techniques compare the system's outputs under testing and the oracle implementation and reveal potential functional bugs with the divergent outputs. Although the techniques can generate the inputs of the systems flexibly, they can only be applied to software systems that have other implementations supporting the same functionality, such as JVM [6], ORM frameworks [40], and SMT solvers [53]. We believe PINOLO and other metamorphic testing approaches are orthogonal to differential testing techniques, which can be applied and strengthened to each other in testing DBMSs.

Grammar fuzzing. Grammar fuzzing is used to generate inputs that satisfy a specific language syntax [8, 11, 19, 19]. It has been widely used in testing many real-world software systems, such as browsers [45], compilers [51], and DBMSs [36, 57]. The generated inputs can always pass the syntax checking of software systems, which avoids the unnecessary enumeration of the inputs in an ill form, improving the effectiveness of generated inputs. Instead of relying on a specific grammar, PINOLO mutates an existing seed query to synthesize an approximate query. Our synthesis process rewrites specific grammatical constructs in the seed query, which ensures the syntactical validity of the synthesized one. We do think it is also promising to utilize existing grammar fuzzing techniques to enumerate initial seed queries automatically [36, 57], which can provide more opportunities for improving the coverage in the DBMS testing.

8 Conclusion

This paper presents PINOLO, an automatic query synthesizer for discovering logical bugs in DBMSs. Given a seed query, PINOLO mutates specific SQL constructs and generates a query that over-approximates or under-approximates the seed query. We posit that the approximation relation provides effective guidance for discovering logical bugs underlying DBMSs. Our experimental results demonstrate the effectiveness of PINOLO. Benefiting from our approximate query synthesis, PINOLO discovers 41 logical bugs in four mature DBMSs. At the time of the submission, 39 bugs have been confirmed by the developers. We hope that the promising results will put forward the study of DBMS testing, further promoting the reliability of database-backed systems.

Acknowledgments

We thank anonymous reviewers and the shepherd for their insightful comments. This work is supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (BK20202001), Natural Science Foundation of China (62272400), Xiamen Youth Innovation Fund (3502Z20206036), and the grant from Huawei.

References

[1] Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. Switchv: Automated sdn switch validation with p4 models. In *Proceedings of the ACM SIGCOMM 2022 Conference*, page 365–379, New York, NY, USA, 2022. ACM.

[2] Jialun Cao, Meiziniu Li, Yeting Li, Ming Wen, Shing-Chi Cheung, and Haiming Chen. Semmt: A semantic-based testing approach for machine translation systems. *ACM Trans. Softw. Eng. Methodol.*, 31(2):1–36, 2022.

[3] Donald D Chamberlin and Raymond F Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET workshop on Data description, access and control*, pages 249–264, 1974.

[4] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. Metamorphic testing: A new approach for generating next test cases. *CoRR*, abs/2002.12543, 2020.

[5] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1):4:1–4:27, 2018.

[6] Yuting Chen, Ting Su, and Zhendong Su. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software Engineering*, pages 1257–1268. IEEE / ACM, 2019.

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

[8] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *Proceedings of 24th USENIX Security Symposium*, pages 193–206. USENIX Association, 2015.

[9] MariaDB Foundation. MariaDB Database. https://mariadb.org/, 2022. [Online; accessed Dec-2022].

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

[11] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 206–215. ACM, 2008.

[12] William E Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, SE-4(4):293–298, 1978.

[13] Free Software Foundation Inc. BNU Bison. https://www.gnu.org/software/bison/, 2022. [Online; accessed Jan-2023].

[14] Google Inc. american fuzzy lop - a security-oriented fuzzer. https://github.com/google/AFL, 2022. [Online; accessed Dec-2022].

[15] Google Inc. Go 1.20 Release Notes. https://tip.golang.org/doc/go1.20, 2022. [Online; accessed Jan-2023].

[16] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226. ACM, 2014.

[17] MaraiaDB. Bug #30249 of MaraiaDB. https://jira.mariadb.org/browse/MDEV-30249, 2022. [Online; accessed Jan-2023].

[18] William M. McKeeman. Differential testing for software. *Digit. Tech. J.*, 10(1):100–107, 1998.

[19] Michaël Mera. Mining constraints for grammar fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 415–418. ACM, 2019.

[20] MySQL. Bug #108937 of MySQL. https://bugs.mysql.com/bug.php?id=108937, 2022. [Online; accessed Jan-2023].

[21] MySQL. Bug #109407 of MySQL. https://bugs.mysql.com/bug.php?id=109407, 2022. [Online; accessed Jan-2023].

[22] MySQL. MySQL Database. https://www.mysql.com/, 2022. [Online; accessed Dec-2022].

[23] MySQL. The MySQL Test Framework. https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_MYSQL_TEST_RUN.html, 2022. [Online; accessed Dec-2022].

[24] OceanBase. A deployer and package manager for OceanBase. https://github.com/oceanbase/obdeploy, 2022. [Online; accessed Jan-2023].

[25] OceanBase. Issue #1100 of OceanBase. https://github.com/oceanbase/oceanbase/issues/1100, 2022. [Online; accessed Jan-2023].

[26] OceanBase. OceanBase Database. https://www.oceanbase.com/, 2022. [Online; accessed Dec-2022].

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

[28] PingCap. Parser - A MySQL Compatible SQL Parser. https://github.com/pingcap/tidb/tree/master/parser, 2022. [Online; accessed Jan-2023].

[29] PingCAP. TiDB Database. https://github.com/pingcap/tidb, 2022. [Online; accessed Dec-2022].

[30] GNU Project. gcov——a Test Coverage Program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2022. [Online; accessed Jan-2023].

[31] qaqcatz. The Complete List of Instantiated Mutators in PINOLO. https://github.com/qaqcatz/impomysql_binary/blob/main/mutators.md, 2022. [Online; accessed Jan-2023].

[32] Mian Qin, Qing Zheng, Jason Lee, Bradley Settlemyer, Fei Wen, Narasimha Reddy, and Paul Gratz. Kvrangedb: Range queries for a hash-based key-value device. *ACM Trans. Storage*, jan 2023.

[33] Manuel Rigger. sqlancer. https://github.com/sqlancer/sqlancer, 2022. [Online; accessed Jan-2023].

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

[35] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.*, 4(OOPSLA):211:1–211:30, 2020.

[36] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation*, pages 667–682. USENIX Association, 2020.

[37] Manuel Rigger and Zhendong Su. Intramorphic testing: A new approach to the test oracle problem. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 128–136. ACM, 2022.

[38] Andreas Seltenreich. SQLsmith: A random SQL query generator. https://github.com/anse1/sqlsmith, 2022. [Online; accessed Dec-2022].

[39] Donald R Slutz. Massive stochastic testing of sql. In *VLDB*, volume 98, pages 618–622. Citeseer, 1998.

[40] Thodoris Sotiropoulos, Stefanos Chaliasos, Vaggelis Atlidakis, Dimitris Mitropoulos, and Diomidis Spinellis. Data-oriented differential testing of object-relational mapping systems. In *43rd IEEE/ACM International Conference on Software Engineering*, pages 1535–1547. IEEE, 2021.

[41] SQLite. How SQLite is Tested. https://www.sqlite.org/testing.html, 2022. [Online; accessed Dec-2022].

[42] TiDB. Issue #40015 of TiDB. https://github.com/pingcap/tidb/issues/40015, 2022. [Online; accessed Jan-2023].

[43] Jianfeng Wang, Tamás Lévai, Zhuojin Li, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. Quadrant: A cloud-deployable NF virtualization platform. In *Proceedings of the 13th Symposium on Cloud Computing*, page 493–509, New York, NY, USA, 2022. ACM.

[44] Jianfeng Wang, Tamás Lévai, Zhuojin Li, Marcos Augusto M. Vieira, Ramesh Govindan, and Barath Raghavan. Galleon: Reshaping the square peg of NFV. *CoRR*, abs/2101.06466, 2021.

[45] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, pages 724–735. IEEE / ACM, 2019.

[46] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. Industry practice of coverage-guided enterprise-level DBMS fuzzing. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*, pages 328–337. IEEE, 2021.

[47] Shuai Wang and Zhendong Su. Metamorphic object insertion for testing object detection systems. In *35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1053–1065. IEEE, 2020.

[48] Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 718–730. ACM, 2020.

[49] Zhiyong Wu, Jie Liang, Mingzhe Wang, Chijin Zhou, and Yu Jiang. Unicorn: detect runtime errors in time-series databases with hybrid input synthesis. In *31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 251–262. ACM, 2022.

[50] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. Metamorphic testing of deep learning compilers. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(1):15:1–15:28, 2022.

[51] Haoran Xu, Shuhui Fan, Yongjun Wang, Zhijian Huang, Hongzuo Xu, and Peidai Xie. Tree2tree structural language modeling for compiler fuzzing. In *Algorithms and Architectures for Parallel Processing - 20th International Conference*, pages 563–578. Springer, 2020.

[52] Yibiao Yang, Yuming Zhou, Hao Sun, Zhendong Su, Zhiqiang Zuo, Lei Xu, and Baowen Xu. Hunting for bugs in code coverage tools via randomized differential testing. In *Proceedings of the 41st International Conference on Software Engineering*, pages 488–498. IEEE / ACM, 2019.

[53] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Fuzzing SMT solvers via two-dimensional input space exploration. In *Proceedings of 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 322–335. ACM, 2021.

[54] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Skeletal approximation enumeration for SMT solver testing. In *Proceedings of 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1141–1153. ACM, 2021.

[55] Yuanyuan Yuan, Shuai Wang, Mingyue Jiang, and Tsong Yueh Chen. Perception matters: Detecting perception failures of VQA models using metamorphic testing. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16908–16917. Computer Vision Foundation / IEEE, 2021.

[56] Andreas Zeller. *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition*. Academic Press, 2009.

[57] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: testing database management systems with language validity and coverage feedback. In *2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 955–970. ACM, 2020.