2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Puppy: Finding Performance Degradation Bugs in DBMSs via Limited-Optimization Plan Construction

Zhiyong Wu*
KLISS, BNRist, School of Software,
Tsinghua University,
Beijing, China

Jie Liang*
KLISS, BNRist, School of Software,
Tsinghua University,
Beijing, China

Jingzhou Fu
KLISS, BNRist, School of Software,
Tsinghua University,
Beijing, China

Mingzhe Wang
KLISS, BNRist, School of Software,
Tsinghua University,
Beijing, China

Yu Jiang†
KLISS, BNRist, School of Software,
Tsinghua University,
Beijing, China

***Abstract*—Database management systems (DBMSs) consistently strive for enhanced performance. For a given query, the optimizer of a DBMS aims to construct an optimal execution plan that incorporates multiple optimization operations. However, the resulting plan may sometimes perform worse than even if no optimizations were applied. This occurs because the interactions between optimizations are complex and some situations might be overlooked in the implementation. We refer to these issues as Performance Degradation Bugs (PDBs). PDBs can result in significant consequences from decreased system efficiency and prolonged query processing times to potential disruptions in critical business operations.**

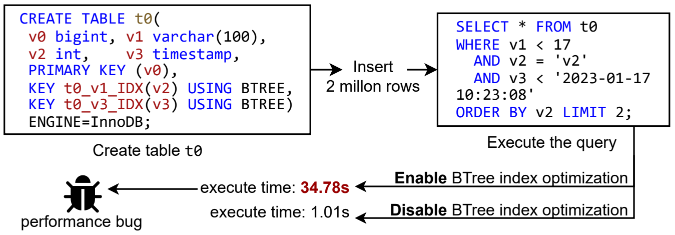
In this paper, we present Puppy, an automated approach for detecting PDBs in DBMSs using limited-optimization plan construction. The key idea is to compare the performance with the plan generated with all optimization operations enabled, against the plan generated with only a subset of optimization operations in the same DBMS. If the response time of the plan with the limited optimization set is shorter than that of the fully optimized plan, it indicates a potential PDB. Specifically, Puppy first generates queries that incorporate multiple optimization sequences, guided by optimization operation sequence coverage. Secondly, Puppy analyzes the query plan and selectively disables specific optimizations to construct the limited optimization plan. We evaluate Puppy on five widely-used DBMSs, namely MySQL, Percona, TiDB, PolarDB, and PostgreSQL against the state-of-the-art DBMS performance testing tools Apollo and Amoeba. More importantly, Puppy reports 62 PDBs, with 54 anomalies confirmed as previously unknown bugs.

I. Introduction

Database Management Systems (DBMSs) serve as the backbone for various applications, ranging from simple web applications to complex, large-scale enterprise systems [36]. Ensuring the efficiency of DBMSs is crucial as it directly influences the responsiveness, scalability, and user satisfaction of dependent applications [6, 8, 37, 9, 33].

To meet the growing performance demands, the *query optimizer* in the DBMS is specifically designed. This component incorporates amounts of sophisticated optimizations for query execution and is considered the most crucial component in the system [17]. The primary objective of the *query optimizer* is to determine the most efficient execution plan for a given query [20]. It aims to minimize the overall time cost of query execution. The plan generated by the query optimizer is typically represented as a tree structure of optimization operations, designed to ensure the optimal performance of the DBMS [16]. This plan is then traversed and eventually transformed into an operation sequence for execution.

However, certain "optimal" optimization sequences calculated by the optimizer may sometimes perform significantly worse than not optimizing at all. This discrepancy arises from the complexity of optimization operation interactions, wherein certain interactions may be overlooked in the implementation, resulting in unintended performance. We refer to these issues as performance degradation bugs (PDBs). For example, Figure 1 presents a performance degradation bug in MySQL, which resulted in a **30x** performance degradation. When executing the query with all optimization operations enabled, MySQL employs BTree Index optimization, resulting in 34.78 seconds to return results. However, when limiting the optimization operations by disabling the BTree Index optimization, MySQL takes only 1.01 seconds to execute the same query and return the same result. This issue arises due to the mismatch between the results of the BTree index optimization and the efficient execution of the ORDER BY clause. The sequence of these two optimizations leads to interference and hinders the overall performance.



```
CREATE TABLE t0(
  v0 bigint, v1 varchar(100),
  v2 int,    v3 timestamp,
  PRIMARY KEY (v0),
  KEY t0_v1_IDX(v2) USING BTREE,
  KEY t0_v3_IDX(v3) USING BTREE)
  ENGINE=InnoDB;
```

```
SELECT * FROM t0
WHERE v1 < 17
  AND v2 = 'v2'
  AND v3 < '2023-01-17
10:23:08'
ORDER BY v2 LIMIT 2;
```

Fig. 1. A performance degradation bug in MySQL 8.0 resulting in a **30x** decrease in response time.

PDBs can result in significant pitfalls as the DBMS fails to achieve the intended performance. These pitfalls can range

*Zhiyong Wu and Jie Liang contributed equally to this work.
†Yu Jiang are the corresponding author.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00045

from decreased system efficiency to potential disruptions in critical business operations. Additionally, it can hinder decision-making processes and may lead to decreased user satisfaction and trust in the DBMS, impacting the overall reputation of the organization relying on the DBMS. Many techniques (e.g., APOLLO [19] and AMOEBA [25]) have been proposed to detect performance bugs in DBMSs. However, they may struggle to effectively detect PDBs caused by unfavorable sequences of optimization operations. Specifically, they typically perform black-box testing on the entire DBMS and cannot delve deeply into the execution details at the optimization level. For example, while APOLLO is capable of identifying performance degradation across different versions of a DBMS, it cannot pinpoint bugs within the query optimizer.

In this paper, we propose PUPPY, which utilizes *limited-optimization plan construction* to detect PDBs in DBMSs. The core idea is to *limit the optimization operations enabled within a plan by modifying certain optimization options.* When a "limited-optimization" plan outperforms the plan with all available optimizations enabled, it indicates a potential PDB. Typically, DBMSs often provide a spectrum of optimization options to control the availability of specific optimization operators. By disabling certain options, we can construct a plan generated with limited optimizations. Nonetheless, implementing this approach still encounters two significant challenges that need to be addressed:

1) Generating SQL queries that incorporate a diverse range of optimization operations. To effectively trigger performance degradation bugs in DBMSs, the generated SQL queries need to include a multitude of optimization operation sequences. However, in the absence of specific guidance, the generated queries may inadvertently contain similar optimization operation sequences.
2) Accurately disabling specific optimization options to construct a plan with limited optimizations. The construction requires understanding the intricacies of the optimization process. Due to the extensive number of optimization options present in the DBMS, identifying and disabling specific options to construct a limited-optimization plan can be a challenging task.

We overcome the two challenges with optimization-guided query synthesis and execution-driven limited-optimization plan construction, respectively. Specifically, we design the optimization operation sequence coverage to steer the SQL query synthesis, thereby enhancing the diversity of the optimization operation sequence in one SQL query. Furthermore, we examine the query plan for the SQL query, gather the options that can impact the utilized optimization operations within the query plan, and subsequently deactivate specific optimizer options based on the information from the execution process analysis. We evaluated PUPPY on five widely-used DBMSs: MySQL, Percona, TiDB, PolarDB, and PostgreSQL. PUPPY reports a total of **62 PDBs**, with 54 performance anomalies have been confirmed as previously unknown bugs. Besides, PUPPY also finds **20 crash bugs**, and 16 of them are confirmed. In addition, we compare PUPPY with the state-of-the-art DBMS validation tools, including both DBMS performance testing tools APOLLO [19] and AMOEBA [25]. The 48-hour result shows that PUPPY detects 26 and 25 more performance bugs, and covered 151,201 and 173,798 more branches than APOLLO and AMOEBA, respectively.

In summary, our paper makes the following contributions:

1) We find that the performance degradation bugs significantly hampered the performance of DBMSs, yet contemporary testing methodologies largely overlook these critical bugs.
2) We propose a test oracle to detect the PDBs in DBMS via limited optimization plan construction. The key idea is to compare the performance of the SQL query with the plan generated by the optimizer when all optimizations are enabled, against the plan generated with only a subset of optimization operations within the same DBMS.
3) We implement our approach in PUPPY. PUPPY reported 62 performance degradation anomalies in five widely used DBMSs, with 54 performance anomalies confirmed as previously unknown bugs.

II. PERFORMANCE DEGRADATION BUGS

Basic Concepts. Typically, a single query can be executed through various ways to retrieve the same set of desired results. An *optimization operation* refers to a predefined algorithm or technique used by the DBMS optimizer to enhance SQL clause performance. For instance, the fundamental approach for data retrieval involves sequentially scanning a table. However, there exist several optimization operations that can enhance this process. For example, if the table is equipped with a B-tree index and the query predicate pertains to the indexed key, the DBMS can leverage the index to expedite retrieval. A *query plan* consists of a sequence of *optimization operations*, which indicates the execution ways [17, 30, 14]. *Query optimization* is a process of optimizing DBMS performance for a query. The instinctive approach to ensuring performance is to select the optimal execution plan that is expected to run the fastest with the selected plan.

The *SQL clause* is a specific command used within an SQL statement to define various aspects of the query's behavior. SQL clauses provide flexibility and control over the desired outcome of a query by specifying conditions, sorting order, grouping criteria, joining multiple tables, and limiting the number of returned results, among other functionalities. Typically, containing more kinds of SQL clauses, the SQL query may trigger more optimization operations of a DBMS.

Definition. A **P**erformance **D**egradation **B**ugs (PDB) refers to the performance of the plan provided by the optimizer with all optimizations enabled is significantly inferior to the plan generated when the partial optimization operation is disabled. In the process of generating plans, an optimizer will try different access paths, join methods, and join orders. In particular, the optimizer initiates the process by scanning each table referenced in the query. Following this, it constructs plans by creating a tree structure of join operations, with each node having two inputs. Ultimately, it assesses multiple potential join sequences to identify the most cost-effective ones. However, the optimization operations in the plan generated by the optimizer may have some intricate scenarios overlooked in the operations' implementation. Executing the plan by combining these optimizations on specific data will cause performance degradation compared to not using optimizations.

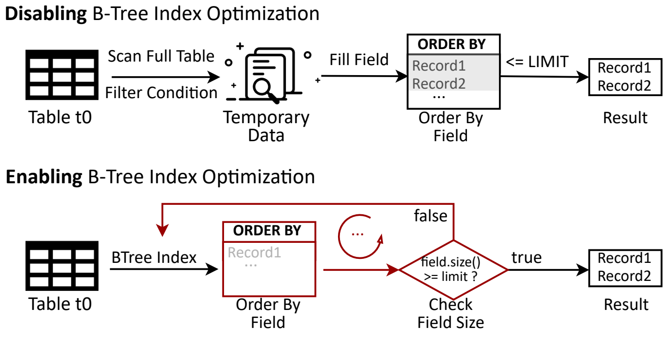


Fig. 2. Execution process with BTree Index Optimization disabled/enabled.

Example. In Section I, we provide an example of a PDB within the optimizer of MySQL. This specific PDB led to a substantial 30x performance degradation in MySQL. In this part, we will conduct a root cause analysis of the example presented in Figure 1 in the plan level. The bug is triggered with the following steps: ① Created an initial empty table with four columns and three indexes, including two BTree indexes (v1 and v3), as well as a primary key index. ② Insert 2 million records into the table t0. ③ Execute the SQL query. MySQL takes 34.78 seconds to return the expected results. ④ Disable the BTree index optimization and re-execute. MySQL takes only 1.01 seconds to execute the same query and return the results.

The process indicates that *BTree index optimization* causes serious performance degradation. Figure 2 shows the execution plans for the SQL query. When all optimization operations are enabled, the optimizer of MySQL generates a plan that combines ORDER BY and BTree Index optimization. Using the BTree Index, MySQL will directly retrieve the eligible records for the ORDER BY field. This retrieval continues until the number of records in the ORDER BY field reaches the number of entries specified in the LIMIT clause. When the size of the result set exceeds the specified value of LIMIT entries, the execution plan can improve the performance through BTree index optimization.

However, if the final result set size is less than the LIMIT entries value, the process will continue to wait for new records until interrupted by other commands. Differently, in the limited-optimization plan when disabling BTree Index optimization, MySQL will have a better performance. Specifically, MySQL first scans and filters all the records based on the conditions without waiting. After that, it sorts the temporary result set for the ORDER BY optimization operators, and finally calculates the results for the number of LIMIT entries.

III. Design of Puppy

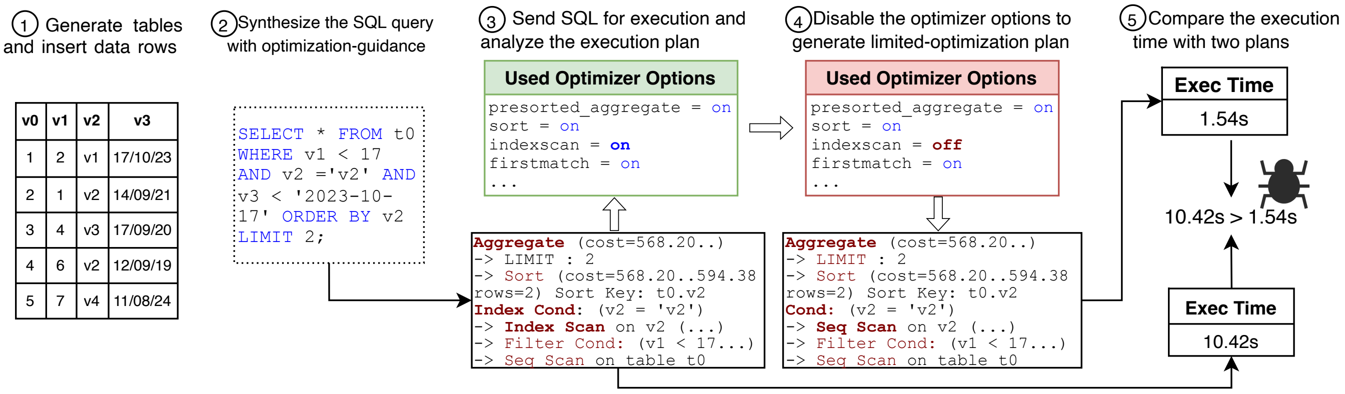
Figure 3 provides a step-by-step illustration of the Puppy's approach. In Step ①, Puppy creates hundreds of basic tables in a fresh database and inserts amounts of initial data. These tables, populated with a random distribution of records ranging from 1 to 100, are designed with diverse features like indexes and foreign keys to provoke complex DBMS optimization behaviors. In Step ②, Puppy randomly picks several tables from this database and synthesizes SQL queries with the optimization guidance. This query can cover new optimization operation sequences and trigger different DBMS behaviors. Moving to Step ③, Puppy sends the synthesized query to DBMS to collect execution time and analyze the execution plan of the SQL query to collect the options that can influence the use of optimization operations. In Step ④, it disables part of the optimizer options to construct a limited-optimization plan. The SQL query will be executed with the limited-optimization plan to collect the execution time again. Finally, in Step ⑤, Puppy compares the executed time of the same query with the original plan and the limited-optimization plan. If the execution time with the limited-optimization plan is significantly less than that of the optimized plan, Puppy detects a PDB in the target DBMS.

There are two major challenges in implementing this approach. First, Puppy needs to generate SQL queries rich in optimization rules that encompass a wide spectrum of optimization combinations during Steps ②. Second, it's crucial to precisely disable certain optimization options, transforming an optimized plan into one with limited optimization for PDB detection in Steps ③ and ④. we address them through optimization-guided SQL synthesis (Section III-A) and execution-driven construction of de-optimized plans(Section III-B), which be detailed in this section. The implementation details of the data generation in Step ① will be described in Section IV.

A. Optimization-Guided SQL Synthesis

Optimization Operation Sequence. A query plan consists of a series of optimization operations. It illustrates the interactions of different optimization operations and the regularity of the combination of optimization operations. Query plans with the same optimization combination sequence trigger similar interactions among optimization operations. To better describe the interaction and abstract the problem, we use the *optimization operation sequence* as a chronological relation of the optimization operation. It is represented as a partially ordered tuple (operation1, operation2, ...). For example, the optimization operation sequence in Step ③ of Figure 3 is (`Seq Scan`, `Index Scan`, `Sort`, `LIMIT`, `Aggregate`). To discover more issues among the interactions of optimization operations within the optimizer, we design optimization operation sequence coverage to guide the SQL synthesis. The optimization operation sequence coverage records all the sequences that have been generated.

SQL Synthesis. With the generated basic data in Step ①, Puppy synthesizes the query with optimization operation


1 Generate tables and insert data rows
2 Synthesize the SQL query with optimization-guidance
3 Send SQL for execution and analyze the execution plan
4 Disable the optimizer options to generate limited-optimization plan
5 Compare the execution time with two plans
Exec Time
1.54s
Exec Time
10.42s
10.42s > 1.54s

v0	v1	v2	v3
1	2	v1	17/10/23
2	1	v2	14/09/21
3	4	v3	17/09/20
4	6	v2	12/09/19
5	7	v4	11/08/24

```
SELECT * FROM t0
WHERE v1 < 17
AND v2 ='v2' AND
v3 < '2023-10-
17' ORDER BY v2
LIMIT 2;
```

Used Optimizer Options

```
presorted_aggregate = on
sort = on
indexscan = on
firstmatch = on
...
```

```
Aggregate (cost=568.20..)
-> LIMIT : 2
-> Sort (cost=568.20..594.38
rows=2) Sort Key: t0.v2
Index Cond: (v2 = 'v2')
-> Index Scan on v2 (...)
-> Filter Cond: (v1 < 17...)
-> Seq Scan on table t0
```

Used Optimizer Options

```
presorted_aggregate = on
sort = on
indexscan = off
firstmatch = on
...
```

```
Aggregate (cost=568.20..)
-> LIMIT : 2
-> Sort (cost=568.20..594.38
rows=2) Sort Key: t0.v2
Cond: (v2 = 'v2')
-> Seq Scan on v2 (...)
-> Filter Cond: (v1 < 17...)
-> Seq Scan on table t0
```

Fig. 3. Approach overview of PUPPY. Step 1: Randomly generate basic tables and insert the initial data rows in the empty databases(Section IV). Step 2: Synthesize the SQL queries with the optimization guidance (Section III-A). The synthesized SQL query will be sent to the target DBMS for execution. Step 3-4: During the execution, PUPPY analyzes the obtained query plan to extract used optimizer options. With calculated used optimizer options, PUPPY disables parts of the used optimization options to construct the limited-optimization plan (Section III-B). Step 5: Compare the execution time with the original plan and limited-optimization plan to identify the PDBs. If the execution time with the limited-optimization plan is shorter than the former, PUPPY detects a PDB.

sequence coverage to trigger more optimization operations. Algorithm 1 outlines the procedure for guiding data evolution and SQL synthesis with the optimization operation sequence coverage. The algorithm requires four inputs: the SQL grammar, the target DBMS, an initially empty Optimization Sequence Set, and the Clause-Optimization Map of the target database.

Algorithm 1: Optimization Operation Sequence Guided Data Evolution and Query Synthesis

```
Input  : SQL Grammar $G$,
         DBMS $D$,
         Operation Sequence Set $O$,
         Clause-Optimization Map $C$
Output: SQL Queries $Q$
1  InitQuery($Q_{temp}$);
2  while $Q_{temp}$ != $null$ do
3      $Q_{temp} \leftarrow$ SQLClauseSynthesis($G$, $C$, $D$);
4      $P \leftarrow$ extractPlan($Q_{temp}$, $D$);
5      $O_p \leftarrow$ getOptimizationSequences($P$);
6      $I_{nterest} \leftarrow$ false;
7      foreach $o \in O_P$ do
8          if $o \notin O$ then
9              $I_{nterest} \leftarrow$ true;
10             addToSequenceSet($O$, $o$);
11         end
12     end
13     if $I_{nterest}$ then
14         addToPoolForExecution($Q_{temp}$, $Q$);
15     end
16 end
17 return $Q$;
```

First, PUPPY selects some tables from the DBMS and initializes and synthesizes a SQL query with stored data and the clause-optimization map following the SQL grammar (Lines 1-3). A "clause-optimization map" is a data structure or mapping that associates specific clauses in an SQL query with optimization operations or strategies that can be applied to those clauses during the query optimization process. For example, it may specify that a certain index optimization should be used for a particular WHERE clause, or it may guide the DBMS to execute join optimization operations in a designated sequence for a JOIN clause. Then, PUPPY analyzes the optimized execution plan and extracts the optimization operation sequences (Lines 4-5). The optimization operation sequence within this plan will be analyzed to identify any novel sequences. The *Optimization Operation Sequence Set* maintains a record of all previously discovered sequences. By checking whether the sequences of this plan exist in this set, we can ascertain the significance of the corresponding query. If it discovers new optimization operation sequences, PUPPY thinks the synthesized SQL query is interesting and adds it to the SQL query pool for execution (Lines 6-15). If the synthesized query can not trigger a new optimization operation sequence, PUPPY extracts the tables used by the query and then changes the stored data in these tables to trigger more behaviors of DBMS in the next iteration (Lines 16-19). For example, let's consider that we generate a query, and PostgreSQL provides its execution plan. Suppose the plan comprises the following operation sequence: Seq Scan, Hash Join, and Seq Scan. Now, if we encounter a new sequence, such as introducing a previously unused operation like BTree Index, the query will be deemed interesting, and it will be added to the query pool for further consideration.

B. Execution-Driven Limited Optimization Plan Construction

After synthesizing amounts of SQL queries, PUPPY then dispatches these queries to the DBMS for execution. During the execution, PUPPY analyzes the default optimization execution plan and constructs limited optimization query plans by disabling some specific optimization options of the DBMS optimizer. Subsequently, the response times of the optimized and limited optimization plans are compared to detect potential optimization degradation issues in the optimizer.

However, selecting appropriate optimization options to disable is challenging. Typically, a DBMS optimizer offers an extensive array of optimization options, which determine the utilization of various optimization operations. Yet, a single SQL query usually triggers only a subset of these optimization operations. If we were to haphazardly deactivate DBMS optimization options, it might become difficult to create a limited-optimization execution plan that is customized to the needs of that particular query.

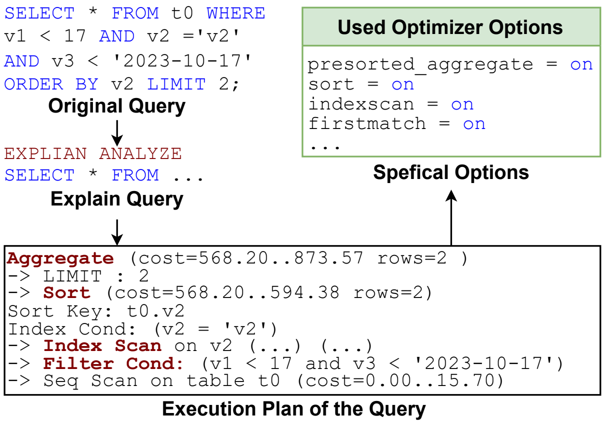
To address the problem, Puppy designs the execution-driven limited optimization plan construction. It has three steps, namely execution process analysis, limited optimization plan construction, and PDB detection. First, Puppy analyzes the query plan of the SQL query and collects the options that can influence the use of optimization operations with the query plan in execution process analysis. Then, Puppy disables parts of the optimizer options with the guidance of results in the first step to construct a limited optimization execution plan. Finally, each limited optimization plan is executed and their response times are collected. For each case in which the response time is significantly less than that of the optimized plan, it serves as a clear indicator of a PDB.

STEP 1: Execution Process Analysis. The execution process analysis consists primarily of two steps. First, it acquires query plans by instrumenting queries with the EXPLAIN statements. For instance, the query plan for the SQL query shown in Figure 1 can be obtained with the command "EXPLAIN SELECT * FROM ...".

TABLE I
Subset of SQL Operations to Options Mappings in PostgreSQL

SQL Operation	Optimization Options
Bitmap Scan	enable_bitmapscan (boolean)
Merge Join	enable_mergejoin (boolean)
Hash Join	enable_hashjoin (boolean)
Index Scan	enable_indexscan (boolean)

Second, Puppy analyzes the acquired query plan to extract *optimization options sequence* using the **Operations-Options Mappings**. These mappings document the SQL optimization options that have the potential to influence the utilization of corresponding optimization operations within the optimizer. A subset of these mapping rules is presented in Table I to showcase their diversity. These mapping rules are constructed by the documentation provided by the DBMS websites.



```
SELECT * FROM t0 WHERE
v1 < 17 AND v2 ='v2'
AND v3 < '2023-10-17'
ORDER BY v2 LIMIT 2;
```

```
EXPLIAN ANALYZE
SELECT * FROM ...
```

```
presorted_aggregate = on
sort = on
indexscan = on
firstmatch = on
...
```

```
Aggregate (cost=568.20..873.57 rows=2 )
-> LIMIT : 2
-> Sort (cost=568.20..594.38 rows=2)
Sort Key: t0.v2
Index Cond: (v2 = 'v2')
-> Index Scan on v2 (...) (...)
-> Filter Cond: (v1 < 17 and v3 < '2023-10-17')
-> Seq Scan on table t0 (cost=0.00..15.70)
```

Fig. 4. An example of the execution process analysis of MySQL.

Figure 4 gives an example of the execution process analysis of MySQL. Puppy first add the 'EXPLAIN ANALYZE' command to obtain the explain query "EXPLAIN ANALYZE SELECT * FROM t0 WHERE v1 < 17 AND v2='v2' AND v3 < '2023-10-17' ORDER BY v2 LIMIT 2". Then, Puppy sends the *explain query* for execution to obtain the execution plan. With the **SQL Operations-Options** mapping as shown in Table I and the query execution plan in Figure 4, Puppy analyzes and calculates the SPECIFICAL optimizer options, which have been used for optimization in the query plan.

STEP 2: Limited-Optimization Plan Construction. With the *optimization option sequence* calculated in execution process analysis, Puppy then disables a specific number (i.e., `limitationCount`) of the optimization options to construct a limited-optimization execution plan for the same query. To ensure the options are disabled differently each time, we use a memorized *option sequence set* to record the options that were chosen to be disabled for plans with the the same optimization option sequence. The value of `limitationCount` is critical for performance anomaly detection. A small value may reduce the testing efficiency, but a too-large value will also miss many true anomalies. The practicable value will be detailed in Section IV.

Algorithm 2: Limited-Optimization Plan Construction

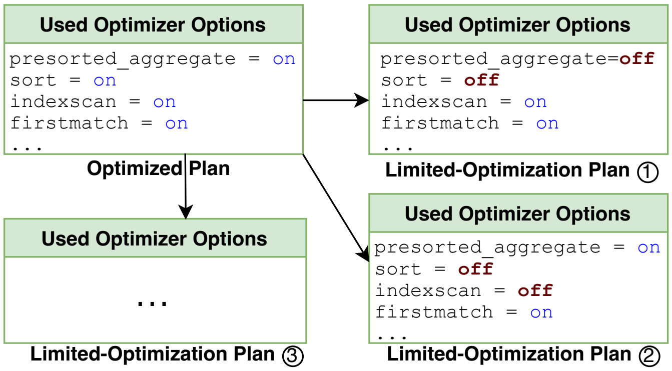
```
Input  : Query Plan $Q$,
         Operation-Options Mappings $M$,
         Operation Sequence Set Map $H$ (option
sequence → list of disabled options set),
         limitationCount $LC$
Output: Optimization Option Set to Be Disabled $S$

$S_{seq}$ ← getOptionSequence($Q$, $M$);
$S_{options}$ ← getOptionSeqSet($H$, $S_{seq}$);
$S_{temp}$ ← genOptionsToDisable($S_{seq}$, $LC$);
while $S_{temp}$ != $null$ do
    if $S_{temp} \notin S_{options}$ then
        $S$ ←$S_{temp}$;
        addToOptionsMap($S_{options}$, $S_{temp}$);
        Break;
    end
    $S_{temp}$ ← genOptionsToDisable($S_{seq}$, $LC$);
end
closeOptimizationoptions($S$);
return $S$ ;
```

Algorithm 2 outlines the complete procedure for constructing a limited-optimization plan with memorized *operation sequence set*. Given an optimized query plan Q, PUPPY initially derives the optimization option sequence S_{seq}. Subsequently, it uses the option sequence set map H to obtain $S_{options}$, which is a list of *disabled option sequence set* for S_{seq}. H maps each *optimization option sequence* to a list of disabled optimization option sets, which are generated for the sequence each time (Lines 1-2). H ensures that the set of optimization options differs each time they are disabled.

Next, PUPPY systematically generates S_{temp}, which is a candidate set of optimization options to be disabled for constructing the limited optimization plan (Line 3). If S_{temp} is found within $S_{options}$, it indicates that S_{temp} has been utilized previously. In such cases, PUPPY will discard S_{temp} and proceed to generate a new one, repeating this process until it obtains a unique optimization option set that is not present in $S_{options}$. Following this, the freshly generated optimization option sequence, S_{temp}, is incorporated into the optimization option set $S_{options}$, and the options contained in it will subsequently be disabled to construct a new limited-optimization plan (Lines 4-11).

Figure 5 provides an illustrative example of the limited optimization plan construction. Using the available optimizer options from the execution process analysis, PUPPY proceeds to disable two of the optimization options, resulting in the creation of different limited optimization plans. Note that the `limitationCount` is 2 in the example, and PUPPY will disable different two optimization options to create many limited optimization plans.



Used Optimizer Options

```
presorted_aggregate = on
sort = on
indexscan = on
firstmatch = on
...
```

Optimized Plan

Used Optimizer Options

```
presorted_aggregate=off
sort = off
indexscan = on
firstmatch = on
...
```

Limited-Optimization Plan ①

Used Optimizer Options

```
presorted_aggregate = on
sort = off
indexscan = off
firstmatch = on
...
```

Limited-Optimization Plan ②

Used Optimizer Options

...

Limited-Optimization Plan ③

Fig. 5. An example of limited optimization plan construction of MySQL.

STEP 3: PDB Detection. Each limited-optimization plan is methodically crafted and executed, and their respective response times are recorded. The objective here is to observe how these limited-optimization plans perform in terms of query execution time when compared to the plan without limitations. If we find that the response time of a limited-optimization plan is notably less than the response time of the unlimited one, it signals a potential PDB. In other words, it suggests that the limited-optimization plan is executing faster than the meticulously optimized plan. This scenario is a cause for concern because it implies that the DBMS's optimizations may not be functioning as intended, and the query execution may be less efficient than expected. Identifying such potential PDBs is a critical aspect of performance testing and database management, as it allows for the detection and subsequent resolution of issues that can impact the overall performance and responsiveness of a database system.

IV. Important Implementation Details

This section explains other implementation details, which we consider significant for the outcome.

Data Generation. To provide the raw material for synthesizing the optimization-related queries, PUPPY first generates the basic data for empty databases, which contain an amount of initial basic tables and lots of stored data. This process unfolds in three stages. Firstly, PUPPY generates a diverse array of tables, each with a randomly chosen quantity (10 to 100) and types of columns. Secondly, it applies two main constraints—indexes and foreign keys—across all tables through a random selection process, ensuring foreign key relationships are correctly established by matching column data types across tables. Finally, PUPPY periodically populates these tables with data, respecting these constraints. This complexity is achieved by leveraging a comprehensive model of SQL grammar, which allows for the random generation of data across various types.

LimitationCount Setting. We find most PDBs by setting the `LimitationCount` in Algorithm 2 (i.e., the number of the disabled optimization options) to a low value. A high value would heavily influence the execution performance of DBMSs, which may cause low testing performance. For example, when the set the `LimitationCount` to 10, PUPPY can only execute the 46% queries than that with 4.

Noise Elimination. The execution of SQL queries can be affected by external factors within the DBMS, potentially affecting performance metrics. For example, repeated executions of a query become faster due to the DBMS caching results after the first run, introducing variability in performance metrics. To enhance efficiency and accuracy in detecting performance degradation bugs within DBMS environments, PUPPY introduces the Noise Elimination Component in PDB reproduction. To mitigate runtime environment impacts, PUPPY collects the SQL queries that trigger anomalies, along with database storage data and optimization settings, to replicate the issue. During the reproduction process, PUPPY first executes the SQL query with an optimized plan. Subsequently, it deactivates the optimization options and reruns the SQL query to neutralize the influence of the cache mechanism. This process is repeated 5 times to eliminate other external runtime influences.

Effort of Adaption. Adapting PUPPY to a new DBMS is a straightforward process requiring minimal effort. It encompasses three primary steps: SQL grammar adaptation, "clause-optimization map" adaption, and adaptation of optimizer options commands. Firstly, PUPPY can autonomously derive the relevant SQL grammar from the provided SQL description. For instance, PostgreSQL employs tools like Bison and Flex to create its SQL parser, allowing the extraction of SQL grammar for query synthesis. Secondly, the clause-optimization map is manually crafted from the specification of DBMSs, which is described in the official document on the website. Most DBMS documentation includes descriptions of optimizer operations and the current SQL grammar, facilitating the adaptation process. It only takes no more than 2 hours and 200 lines of code to build the "clause-optimization map" for MySQL. Finally, DBMS documentation often provides SQL commands to configure optimization options directly. For example, MySQL has more than 30 optimization options related to performance issues, as detailed in sources like [47, 32]. These options can be seamlessly incorporated into Puppy for efficient adaptation.

V. Evaluation

We evaluate Puppy in terms of its ability to discover the performance degradation bugs, as well as its efficiency in triggering the optimization operation combinations of DBMS. Our evaluation aims to answer the following questions:

- **RQ1:** Can Puppy discover performance degradation bugs in widely-used DBMSs?
- **RQ2:** How does Puppy's performance compare to other DBMS testing techniques?
- **RQ3:** How important is the optimization guided algorithms in identifying PDB?

A. Evaluation Setup

To evaluate the generality and efficiency of Puppy, we perform experiments on five popular open-source DBMSs, namely MySQL [1, 47], Percona [7], TiDB [35, 15], PolarDB [3, 2], and PostgreSQL [4, 29], which are widely used in industry. We conducted experiments on a machine running 64-bit Ubuntu 20.04 with an AMD EPYC 7742 Processor @ 2.25 GHz, 128 cores, and 504 GiB of main memory. All DBMSs were tested using docker containers that were downloaded directly from their websites. We allocated 5 CPU cores and 40 GiB of RAM for the docker containers of each DBMS testing.

B. DBMS Performance Degradation Bug Detection

We applied Puppy to MySQL, Percona, TiDB, PolarDB, and PostgreSQL to test performance degradation bugs for six weeks. These evaluated DBMSs are widely used and have been tested for decades. Nevertheless, Puppy still performed well. 54 PDBs and 16 crash bugs are confirmed as previously unknown bugs in these DBMSs.

Overall Results. Table II shows the statistics of the bugs detected with Puppy in six-week continuous DBMS testing. It shows Puppy identified a total of 62 unique PDBs. Out of these, 54 PDBs have been confirmed, while 8 anomalies are currently undergoing validation by the developers, owing to the intricacies of the DBMS. Among the confirmed PDBs, 23 have been fixed at the time of writing this paper, and we have received expressions of gratitude from the developers. Detecting performance degradation bugs is challenging because they cannot be directly identified by estimating the response time of an SQL query. Furthermore, these bugs are intertwined with the optimization operation combinations of the optimizer, necessitating SQL queries with various clauses to trigger them.

Besides PDBs, Puppy also detected 20 crash bugs in the tested DBMS, 16 of them have been confirmed as previously unknown crash bugs and 12 of them have been fixed. These bugs are discovered because Puppy generates queries that encompass a variety of distinct optimization operation combinations. Thus Puppy covers deep behaviors of the DBMSs and triggers these crashes.

TABLE II
Number of confirmed and fixed unique bugs detected by Puppy.

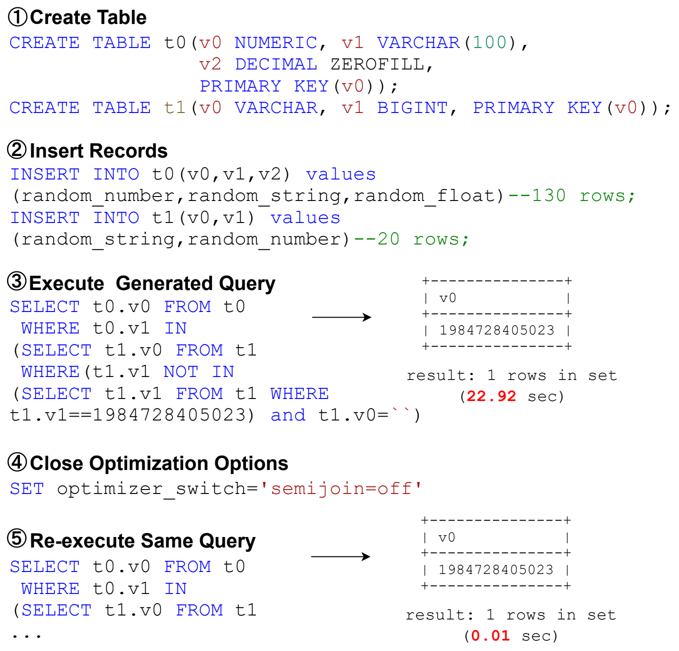
	PDB			Crash		
DBMS	Reported	Confirmed	Fixed	Reported	Confirmed	Fixed
MySQL	15	13	6	5	4	4
Percona	20	17	8	8	6	4
TiDB	9	9	5	3	2	2
PolarDB	12	11	4	3	3	1
PostgreSQL	6	4	1	1	1	1
Total	62	54	23	20	16	12

Bug Severity. Based on the analysis conducted by DBMS developers, the PDBs identified by Puppy are associated with 309 optimization operation combinations within the optimizers of the tested DBMSs. While these bugs are elusive to detect, their consequences result in significant performance degradation of the DBMSs. Among 62 identified PDBs, 41 bugs cause over 500% performance degradation. As of the time of writing this paper, a total of 23 bugs have been fixed. In our communications with the developers, they expressed their astonishment at the extent of performance degradation caused by these bugs and their keen interest in our methods for detecting them. As one developer of MySQL said,

> *"We greatly appreciate your raising this issue as it has brought to light an oversight in our optimization design! We will promptly address and rectify this design concern."*

Case Study: *A limited optimization bug resulting in a 220-fold performance decline caused by semi-join and index_scan in Percona.* Figure 6 illustrates the complete process of triggering the performance degradation bug. In Step ①, Puppy creates two complex tables incorporating primiary_key constraints. With these constraints in place, Percona optimizes data retrieval by scanning the primiary_key column using the index_scan operation. In Step ②, Puppy populates these tables with records adhering to the defined constraints. Subsequently, Puppy formulates a SQL query containing intricate clauses that necessitate scanning specific records from table t0, thereby invoking Percona's semi-join optimization. Notably, Percona requires 22.92 seconds to execute this SQL query in Step ③. However, in Step ④, upon disabling the semi-join optimization, the identical SQL query is executed by Percona in a mere 0.01 seconds. This noteworthy contrast underscores the substantial performance decline induced by the semi-join optimization, resulting in a 200-fold slowdown compared to its absence during execution.

The root cause of the bug. In Percona, the semi-join represents a classical technique for joining operations, where rows



①Create Table

```
CREATE TABLE t0(v0 NUMERIC, v1 VARCHAR(100),
                v2 DECIMAL ZEROFILL,
                PRIMARY KEY(v0));
CREATE TABLE t1(v0 VARCHAR, v1 BIGINT, PRIMARY KEY(v0));
```

②Insert Records

```
INSERT INTO t0(v0,v1,v2) values
(random_number,random_string,random_float)--130 rows;
INSERT INTO t1(v0,v1) values
(random_string,random_number)--20 rows;
```

③Execute Generated Query

```
SELECT t0.v0 FROM t0
 WHERE t0.v1 IN
(SELECT t1.v0 FROM t1
 WHERE(t1.v1 NOT IN
(SELECT t1.v1 FROM t1 WHERE
t1.v1==1984728405023) and t1.v0=``)
```

```
+---------------+
| v0            |
+---------------+
| 1984728405023 |
+---------------+

result: 1 rows in set
     (22.92 sec)
```

④Close Optimization Options

```
SET optimizer_switch='semijoin=off'
```

⑤Re-execute Same Query

```
SELECT t0.v0 FROM t0
 WHERE t0.v1 IN
(SELECT t1.v0 FROM t1
...
```

```
+---------------+
| v0            |
+---------------+
| 1984728405023 |
+---------------+

result: 1 rows in set
     (0.01 sec)
```

Fig. 6. The steps to trigger a performance degradation bug in Percona.

from one table are filtered based on matching rows in another table. Unlike regular joins, which yield a blend of columns from both tables, a semi-join exclusively returns rows from the first table if they meet a specified condition with the second table. Usually, employing the semi-join optimization is expected to yield better performance than not using optimization.

In this case, Percona does not deal with the situation that the two IN optimizations and semi-join optimization are both enabled. Specifically, in Step ③, three tables need to be optimized with semi-join. During the execution, the IN expression of the NOT IN statement first calculates the zero rows that meet the condition of the next joined table. The semi-join optimization hangs until the NOT IN statement finally returns the real result. Consequently, semi-join and optimization led to a staggering 220-fold performance decline when applied with IN expression optimization in comparison to its absence. To fix the bug, the developers of MySQL add codes for semi-join optimization to handle the combination with nested IN optimizations.

Why the bug was only discovered by PUPPY*?* We also test Percona with APOLLO and AMOEBA but they do not find this bug. This bug is tied to enabling and disabling semi-join optimization within the Percona, presenting a challenge for detection using traditional DBMS testing methods. For example, AMOEBA identifies bugs by generating equivalent SQL queries and comparing their performance. However, since these queries do not impact the DBMS optimization operations, AMOEBA is hard to detect this specific bug. Similarly, APOLLO identifies performance regression bugs through a comparison of response times between DBMS versions with identical operation options. Consequently, this method is unable to reveal this particular bug.

In contrast, PUPPY utilizes a distinctive methodology for bug detection. It generates the executed SQL query using optimization-guided SQL synthesis, meticulously analyzes the query plan and optimization operations to construct a de-optimized execution plan, and subsequently compares the response time with the optimized plan. This process enables PUPPY to identify performance degradation issues effectively and it successfully detected the performance degradation bug by disabling the semi-join optimization of Percona. In summary, these results unequivocally demonstrate that PUPPY excels at uncovering previously unknown correctness and performance bugs, providing a robust response to **RQ1**.

C. Compare with Other Techniques

To evaluate the effectiveness of our test oracle, we conducted a comparison experiment between PUPPY with APOLLO and AMOEBA, two state-of-the-art DBMS performance testing methods. Note that AMOEBA only provides an artifact to test PostgreSQL and source code is currently not available, we implement it following its paper [26] based on the equivalent SQL query construction on MySQL, TiDB, Percona, and PolarDB. We ran the testing tools on each DBMS for 48 hours and recorded the number of detected performance bugs and covered branches as the metric.

TABLE III
NUMBER OF PERFORMANCE BUGS DETECTED BY APOLLO, AMOEBA AND PUPPY IN 48 HOURS.

DBMS	APOLLO	AMOEBA	PUPPY
MySQL	1	1	9
Percona	1	1	8
TiDB	0	1	5
PolarDB	2	1	6
PostgreSQL	0	1	2
Total	4	5	30

PUPPY outperforms other performance testing techniques in finding performance anomalies. Table III displays the number of performance bugs detected by each tool. In 48 hours, PUPPY detected a total of 30 performance degradation bugs on MySQL, Percona, TiDB, PolarDB, and PostgreSQL in 48 hours, while APOLLO and AMOEBA only found 4 and 5 performance bugs, respectively. Compared to APOLLO and AMOEBA, PUPPY detected 26 and 25 more performance anomalies, respectively.

One of the main reasons that PUPPY discovered more bugs is that PUPPY can cover more branches and trigger more behaviors of DBMSs than APOLLO and AMOEBA. Table IV shows the number of branches covered by each technique in 48 hours. It shows that PUPPY covers more branches in 48 hours when compared to APOLLO and AMOEBA. Specifically, compared to the other two performance testing methods, PUPPY covered a total of 151,201 and 173,798 more branches than APOLLO and AMOEBA, respectively. The ability to cover more branches can be attributed to PUPPY's generation of SQL

TABLE IV
Number of branches covered by three tools in 48 hours.

DBMS	Apollo	Amoeba	Puppy
MySQL	45,532	43,194	74,658
Percona	49,523	32,932	77,863
TiDB	33,532	29,345	54,924
PolarDB	53,923	44,563	69,837
PostgreSQL	43,556	53,435	99,985
Total	226,066	203,469	377,267

queries guided by optimization operation sequence coverage. This approach facilitates the activation of a broader range of optimization strategy sequences and the exploration of a wider spectrum of optimizer behaviors in DBMSs. In summary, the results indicate that Puppy can detect more performance bugs and cover more branches than Apollo and Amoeba, which adequately answers **RQ2**.

D. Efficiency of the Optimization Guided Algorithm

Section III-A highlights that Puppy proposes the optimization operation sequence coverage to guide the SQL clause synthesis, which increases the number of unique optimization strategy sequences of the execution plans for the SQL queries.

To investigate the effectiveness of the optimization operation sequence coverage guidance on Puppy to discover the performance degradation issues, we implement Puppy-, which disables the optimization operation sequence coverage guidance algorithms. Note that Puppy- generates the SQL query by synthesizing the SQL clause randomly. We compare Puppy against Puppy- on MySQL, Percona, TiDB, PolarDB, and PostgreSQL for 48 hours and collect the number of triggered PDBs as well as the number of optimization operation sequences as the metric.

TABLE V
The number of performance bugs and optimization operation sequence number triggered by Puppy and Puppy- in 48 hours

	PDBs		Optimization Operation Sequences		
	Puppy-	Puppy	Puppy-	Puppy	Improvement
MySQL	6	9	364,853	539,048	47.74%↑
Percona	4	8	394,836	574,938	45.61%↑
TiDB	3	5	422,192	736,372	74.42%↑
PolarDB	5	6	382,193	628,228	64.37%↑
PostgreSQL	0	2	563,182	826,927	46.83%↑
Total	18	30	2,127,256	3,305,513	55.39%↑

Table V shows the number of detected PDBs as well as the number of optimization strategy sequences triggered by Puppy and Puppy- on MySQL, Percona, TiDB, PolarDB, and PostgreSQL after testing for 48 hours. From the left part of the table, we can see that Puppy discovered 12 more PDBs than Puppy- on five DBMSs. Specifically, Puppy detected 3, 4, 2, 1, and 2 more PDBs on MySQL, Percona, TiDB, PolarDB, and PostgreSQL in 48 hours, respectively. The reason behind the enhancement in the number of detected bugs can be elucidated by the impact on the quantity of activated optimization operation sequences in the execution plan.

From the right part of the table, we can also notice that Puppy total achieves a total of 55.39% improvement in the number of triggered optimization operation sequences on five DBMS. Specifically, Puppy triggered 47.74%, 45.61%, 74.42%, 64.37% and 46.83% more optimization operation sequences than Puppy-on MySQL, Percona, TiDB, PolarDB, and PostgreSQL, respectively. The result is reasonable because our design of optimization-guided SQL synthesis was intended to stimulate a broader array of optimization operation sequences within the optimizer. Without optimization guiding, Puppy- only synthesizes new SQL queries by randomly selecting SQL clauses. Therefore, most of the inputs generated by Puppy- triggered repetitive optimization operation sequences and failed to explore a broader state space within the DBMS. In contrast, Puppy synthesizes SQL queries with the optimization operation sequences coverage guidance. Therefore, Puppy can trigger more optimization operation sequences and detect more performance bugs when compared to Puppy-.

In summary, the experiments have demonstrated that optimizing operation sequence coverage contributes to an increase in the number of optimized sequences covered in generated queries and the discovery of performance issues. This provides a comprehensive answer to **RQ3**.

VI. Discussion

False Positives Analysis. Puppy identifies performance anomalies by comparing the execution response time between the limited optimization plan and the unlimited plan. When this disparity exceeds a specified **degradation margin**, Puppy flags it as a PDB. *We report the detected performance anomalies to developers, and only the confirmed anomalies by developers are considered real bugs, the others are considered false positives.* The degradation margin parameter significantly influences the occurrence of false positives. A low degradation margin value can lead to a high number of false positives, while an excessively high degradation margin may cause genuine anomalies to go undetected.

TABLE VI
Number of reported anomalies, false positives, and true positives of Puppy on various degradation margin values.

Degradation Margin	Reported	False Positives	True Positives
1.2	20	10	10
1.5	11	2	9
2	8	1	7
3	6	0	6
4	6	0	6
5	5	0	5

To find a pragmatic degradation margin, We conduct the experiment that runs Puppy on MySQL with different degradation margin values, ranging from 1.2 to 5. We established a minimum threshold of 1.2 for identifying performance anomalies, based on our communication with DBMS developers during the bug-reporting process. Table VI shows the number of reported anomalies, false positives, and true positives of Puppy in 48 hours. According to the experimental data, as the degradation margin increases from 1.2 to 5, false positives continuously decrease. However, due to strict threshold settings, the tool's reported true positives also decreased. In the table, we observe that when the threshold is raised from 1.2 to 1.5, false positives decrease by 80%, while true positives decrease by only 1. In cases where human resources are abundant, a lower threshold can be set, allowing for manual verification even with a higher occurrence of false positives. However, in our experiments, to minimize false positives while capturing more genuine issues, we opted for a threshold of 1.5.

Compare With Other DBMS Testing Tools. In Section V-C, we compare Puppy with Apollo and Amoeba to evaluate its ability to detect performance bugs. To further evaluate the performance of Puppy, we also compare Puppy with SQLancer, SQLsmith, and Squirrel, which are widely used in industry. Table VII shows the number of detected bugs by each tool in 48 hours. It shows that Puppy outperforms SQLancer and SQLsmith in detecting bugs. Specifically, Puppy detect detected a total of 35 bugs (including 30 performance bugs and 5 crash bugs) in 48 hours, while SQLancer, SQLsmith and Squirrel only detected 29, 31, and 30 bugs in total.

TABLE VII
Detected bugs by SQLancer, SQLsmith and Puppy in 48 hours.

DBMS	SQLancer	SQLsmith	Squirrel	Puppy
MySQL	2	1	2	11
Percona	2	1	1	9
TiDB	1	0	0	5
PolarDB	1	1	2	8
PostgreSQL	0	1	0	2
Total	6	4	5	35

VII. Related Work

A. DBMS Fuzzing

Fuzzing is an automated software testing technique, which generates random data as program inputs. It has been widely adopted in practice for finding bugs in many critical areas, including operating systems [43], networking protocols [28, 27], third-part libraries [34, 23], and DBMS. DBMS Fuzzing is an effective technique for finding bugs in DBMSs. It could be divided into generation-based fuzzing or mutation-based fuzzing. Generation-based fuzzing generates SQL queries based on pre-defined syntax or grammar models. SQLsmith [41] generates queries based on built-in code that embeds the AST generation rules for the target DBMS. SQLancer [40, 39, 38] aims to find logic bugs and it generates queries based on the test oracle it builds.

Mutation-based fuzzers mutate existing queries to produce new queries. Squirrel [51], Lego [22] and Ratel [46] mutates queries based on their AST structure. DynSQL [18] captures DBMS state information for each statement to incrementally generate complex and valid SQL queries. Griffin [13] introduces a grammar-free way to reshuffle existing statements from different queries and then repairs their semantic correctness by tracking database schema. Sedar [12] import the LLM help transfer the existing test cases for fuzzing. QPG [5] gradually mutates DDL and DML statements to change database states, aiming to cover more unique query plans to cover more DBMS logic. Unicorn [48] detects the implicit exceptions in the time-series database with hybrid input synthesis. WingFuzz [24] implements continuous fuzzing for DBMS.

Puppy could be regarded as a generation-based fuzzer. Different from other generation-based fuzzers, Puppy generates SQL queries that incorporate a diverse range of optimization strategies. It utilizes optimization operation sequence coverage to guide the process of inserting different SQL clauses into the queries. Compared to QPG which utilizes plan-based guidance, using the sequence of optimization operations within the plan offers a finer-grained form of guidance. This enables Puppy to achieve a more precise coverage of various optimization operations.

B. DBMS Performance Testing

The enduring goal within the realm of the DBMS has always been to attain exceptional performance. Traditional performance testing like TPC-H [45] benchmarks by executing DBMS under pre-defined workloads. Apollo [19] utilizes differential testing on multiple versions to test regression testing. Amoeba [25] generates equivalent queries and compares their response time to find performance issues.

Puppy is designed to detect PDBs, essentially a facet of performance testing. Unlike conventional approaches that establish performance baselines through pre-defined values (e.g., TPC-H), comparisons with different versions of a DBMS like Apollo, or equivalent query execution values as seen with Amoeba, Puppy takes a unique approach. It selectively disables specific optimizations within a DBMS and evaluates whether the same query runs unexpectedly slowly.

C. DBMS Configuration Tuning

DBMS configuration tuning is a practice focused on enhancing the performance and efficiency of a DBMS by fine-tuning various configurations. Both DB2 of IBM and Oracle have implemented a heuristic-based approach for memory allocation management, which relies on performance measurements [42, 10]. Oracle and Microsoft provide SQL analyzer tools that assess the potential performance impact of DBMS modifications for configuration adjusting [50, 31]. Other research endeavors aim to automate the adjustment of DBMS configurations. They typically execute a workload in the target DBMS, make configuration changes, and then rerun the same workload to assess whether there has been a performance improvement [11, 49, 44, 21]. For example, iTuned [11] employs a “cycle stealing” strategy to continually evaluate the influence of adjusting specific configurations on performance. OtterTune [44] leverages a mix of supervised and unsupervised machine learning techniques to identify critical configurations, adapt to new workloads, and recommend optimal settings.

Puppy is designed to uncover performance degradation bugs rather than seeking an optimal configuration. It achieves this by selectively disabling certain well-established optimization strategies to identify potential performance issues.

VIII. Conclusion

We propose Puppy to detect performance degradation bugs. We find that some specific optimization operation sequences in the optimizer may cause serious performance degradation of DBMSs. However, current works usually test DBMSs without changing the optimization operation sequence of the SQL query, which might ignore the performance degradation bugs. Consequently, we design an automated approach to detect performance degradation bugs using limited-optimization plan construction and implement it in Puppy. We evaluate Puppy on widely-used DBMSs and find 62 previously unknown performance bugs and 20 crash bugs.

Acknowledgment

We appreciate the valuable comments provided by the reviewers. This research is partly sponsored by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62302256, 92167101, 62021002), and Chinese Postdoctoral Science Foundation(2023M731953).

References

[1] Mysql. https://www.mysql.com/. Accessed: August 19, 2024.

[2] Polardb documentation. https://www.alibabacloud.com/help/en. Accessed: August 19, 2024.

[3] Polardb github. https://github.com/polardb/polardbx-sql. Accessed: August 19, 2024.

[4] Postgresql. https://www.postgresql.org/. Accessed: August 19, 2024.

[5] Ba, J., and Rigger, M. Testing database engines via query plan guidance. In *Proceedings of International Conference on Software Engineering (ICSE)* (2023).

[6] Bannister, A. Sqlite patches use-after-free bug that left apps open to code execution, denial-of-service exploits. https://portswigger.net/daily-swig/sqlite-patches-use-after-free-bug-that-left-apps-open-to-code-execution-denial-of-service-exploits, 5 2021. Accessed: August 19, 2024.

[7] Blåudd, A. M. Percona website. https://www.percona.com/. Accessed: August 19, 2024.

[8] Cimpanu, C. Google chrome impacted by new magellan 2.0 vulnerabilities. https://www.zdnet.com/article/google-chrome-impacted-by-new-magellan-2-0-vulnerabilities/, 12 2019. Accessed: August 19, 2024.

[9] Community, P. Postgresql bug list. https://www.postgresql.org/. Accessed: August 19, 2024.

[10] Dias, K., Ramacher, M., Shaft, U., Venkataramani, V., and Wood, G. Automatic performance diagnosis and tuning in oracle. In *CIDR* (2005), pp. 84–94.

[11] Duan, S., Thummala, V., and Babu, S. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment 2*, 1 (2009), 1246–1257.

[12] Fu, J., Liang, J., Wu, Z., and Jiang, Y. Sedar: Obtaining high-quality seeds for dbms fuzzing via cross-dbms sql transfer. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (2024), pp. 1–12.

[13] Fu, J., Liang, J., Wu, Z., Wang, M., and Jiang, Y. Griffin: Grammar-free dbms fuzzing. In *Conference on Automated Software Engineering (ASE'22)* (2022).

[14] Group, T. P. G. D. Using explain. https://www.postgresql.org/docs/current/using-explain.html, 1 2024. Accessed: August 19, 2024.

[15] Huang, D., Liu, Q., Cui, Q., Fang, Z., Ma, X., Xu, F., Shen, L., Tang, L., Zhou, Y., Huang, M., et al. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment 13*, 12 (2020), 3072–3084.

[16] Ioannidis, Y. E. Query optimization. *ACM Computing Surveys (CSUR) 28*, 1 (1996), 121–123.

[17] Jarke, M., and Koch, J. Query optimization in database systems. *ACM Computing surveys (CsUR) 16*, 2 (1984), 111–152.

[18] Jiang, Z.-M., Bai, J.-J., and Su, Z. Dynsql: Stateful fuzzing for database management systems with complex and valid sql query generation, 2023.

[19] Jung, J., Hu, H., Arulraj, J., Kim, T., and Kang, W. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems (to appear). In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)* (Tokyo, Japan, Aug. 2020).

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

[21] Li, G., Zhou, X., Li, S., and Gao, B. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment 12*, 12 (2019), 2118–2130.

[22] Liang, J., Chen, Y., Wu, Z., Fu, J., Wang, M., Jiang, Y., Huang, X., Chen, T., Wang, J., and Li, J. Sequence-oriented dbms fuzzing. In *2023 IEEE International Conference on Data Engineering (ICDE)*, IEEE.

[23] Liang, J., Wang, M., Zhou, C., Wu, Z., Jiang, Y., Liu, J., Liu, Z., and Sun, J. Pata: Fuzzing with path aware taint analysis. In *2022 2022 IEEE Symposium on Security and Privacy (SP)(SP). IEEE Computer Society, Los Alamitos, CA, USA. 154ś170* (2022).

[24] Liang, J., Wu, Z., Fu, J., Bai, Y., Zhang, Q., and Jiang, Y. {WingFuzz}: Implementing continuous fuzzing for {DBMSs}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)* (2024), pp. 479–492.

[25] Liu, X., Zhou, Q., Arulraj, J., and Orso, A. Automatic detection of performance bugs in database systems using equivalent queries.

[26] Liu, X., Zhou, Q., Arulraj, J., and Orso, A. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 225–236.

[27] Luo, Z., Zuo, F., Jiang, Y., Gao, J., Jiao, X., and Sun, J. Polar: Function code aware fuzz testing of ICS protocol. *ACM Trans. Embed. Comput. Syst. 18*, 5s (2019), 93:1–93:22.

[28] Luo, Z., Zuo, F., Shen, Y., Jiao, X., Chang, W., and Jiang, Y. Ics protocol fuzzing: coverage guided packet crack and generation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)* (2020), IEEE, pp. 1–6.

[29] Momjian, B. *PostgreSQL: introduction and concepts*, vol. 192. Addison-Wesley New York, 2001.

[30] Understanding the query execution plan. https://dev.mysql.com/doc/refman/8.0/en/execution-plan-information.html, 1 2024. Accessed: August 19, 2024.

[31] Narayanan, D., Thereska, E., and Ailamaki, A. Continuous resource monitoring for self-predicting dbms. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (2005), IEEE, pp. 239–248.

[32] Oracle. Mysql explain manual. https://dev.mysql.com/doc/refman/8.0/en/using-explain.html. Accessed: August 19, 2024.

[33] Oracle. Mysql bug list. https://bugs.mysql.com/, 1 2014. Accessed: August 19, 2024.

[34] Continuous fuzzing for open source software. https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html, 2016. Accessed: August 19, 2024.

[35] PingCAP. Tidb. https://github.com/pingcap/tidb. Accessed: August 19, 2024.

[36] Ramakrishnan, R., Gehrke, J., and Gehrke, J. *Database management systems*, vol. 3. McGraw-Hill New York, 2003.

[37] Rigger, M. Bugs found in database management systems. https://www.manuelrigger.at/dbms-bugs, 2024. Accessed: August 19, 2024.

[38] Rigger, M., and Su, Z. Finding bugs in database systems via query partitioning. pacmpl 4 (oopsla)(nov 2020).

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

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

[41] Seltenreich, A., Tang, B., and Mullender, S. Sqlsmith: a random sql query generator.

[42] Storm, A. J., Garcia-Arellano, C., Lightstone, S. S., Diao, Y., and Surendra, M. Adaptive self-tuning memory in db2. In *Proceedings of the 32nd international conference on Very large data bases* (2006), pp. 1081–1092.

[43] Sun, H., Shen, Y., Wang, C., Liu, J., Jiang, Y., Chen, T., and Cui, A. HEALER: relation learning guided kernel fuzzing. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021* (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 344–358.

[44] Van Aken, D., Pavlo, A., Gordon, G. J., and Zhang, B. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data* (2017), pp. 1009–1024.

[45] Vershinin, I., and Mustafina, A. Performance analysis of postgresql, mysql, microsoft sql server systems based on tpc-h tests. In *2021 International Russian Automation Conference (RusAutoCon)* (2021), IEEE, pp. 683–687.

[46] Wang, M., Wu, Z., Xu, X., Liang, J., Zhou, C., Zhang, H., and Jiang, Y. Industry practice of coverage-guided enterprise-level dbms fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (2021), IEEE, pp. 328–337.

[47] Widenius, M., Axmark, D., and Arno, K. *MySQL reference manual: documentation from the source.* " O'Reilly Media, Inc.", 2002.

[48] Wu, Z., Liang, J., Wang, M., Zhou, C., and Jiang, Y. Unicorn: Detect runtime errors in time-series databases with hybrid input synthesis. In *Symposium on Software Testing and Analysis (ISSTA'22)* (2022).

[49] Xi, B., Liu, Z., Raghavachari, M., Xia, C. H., and Zhang, L. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th international conference on World Wide Web* (2004), pp. 287–296.

[50] Yagoub, K., Belknap, P., Dageville, B., Dias, K., Joshi, S., and Yu, H. Oracle's sql performance analyzer. *IEEE Data Eng. Bull. 31*, 1 (2008), 51–58.

[51] Zhong, R., Chen, Y., Hu, H., Zhang, H., Lee, W., and Wu, D. Squirrel: Testing database management systems with language validity and coverage feedback. In *The ACM Conference on Computer and Communications Security (CCS), 2020* (2020).