Mozi: Discovering DBMS Bugs via Configuration-Based Equivalent Transformation

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

Zhiyong Wu*
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

Chengnian Sun
Cheriton School of Computer Science,
University of Waterloo
Waterloo, Canada

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

ABSTRACT

Testing database management systems (DBMSs) is a complex task. Traditional approaches, such as metamorphic testing, need a precise comprehension of the SQL specification to create diverse inputs with equivalent semantics. The vagueness and intricacy of the SQL specification make it challenging to accurately model query semantics, thereby posing difficulties in testing the correctness and performance of DBMSs. To address this, we propose Mozi, a framework that finds DBMS bugs via configuration-based equivalent transformation. The key idea behind Mozi is to compare the results of equivalent DBMSs with different configurations, rather than between semantically equivalent queries. The framework involves analyzing the query plan, changing configurations to transform the DBMS to an equivalent one, and re-executing the query to compare the results using various test oracles. For example, detecting differences in query results indicates correctness bugs, while observing faster execution times on the optimization-closed DBMS suggests performance bugs.

We demonstrate the effectiveness of Mozi by evaluating it on four widely used DBMSs, namely MySQL, MariaDB, Clickhouse, and PostgreSQL. In the continuous testing, Mozi found a total of 101 previously unknown bugs, including 49 correctness and 52 performance bugs in four DBMSs. Among them, 90 bugs are confirmed and 57 bugs have been fixed. In addition, Mozi can be extended to other DBMS fuzzers for testing various types of bugs. With Mozi, testing DBMSs becomes simpler and more effective, potentially saving time and effort that would otherwise be spent on precisely modeling SQL specifications for testing purposes.

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

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

DBMS Testing, Configuration, Test Oracle

ACM Reference Format:
Jie Liang, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Chengnian Sun, and Yu Jiang. 2024. Mozi: Discovering DBMS Bugs via Configuration-Based Equivalent Transformation. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639112

1 INTRODUCTION

Database Management Systems (DBMSs) serve as the backbone for various applications, ranging from simple web applications to complex, large-scale enterprise systems [11, 15, 58]. It is crucial to ensure the correctness, reliability, and performance of DBMSs [4, 8, 10, 40, 44]. Metamorphic testing is a prevalent approach to testing DBMS, which focuses on building metamorphic relations [5, 6, 32, 46, 47, 49]. These relations represent mathematical relationships between the input and output of a software system, which should hold even when the input undergoes certain changes or transformations. One common way of building metamorphic relations for DBMSs is by creating different SQL queries with equivalent semantics and comparing their outputs. For instance, consider the SQL queries "SELECT 1+1" and "SELECT 2." Although they have different syntactic representations, their semantics are equivalent, as both yield the same result.

To identify metamorphic relations, a precise comprehension of the SQL specification is necessary [17, 18, 36]. *However, the inherent vagueness and intricacy of the SQL specification make accurately modeling the semantics of queries difficult.* Moreover, it often requires a high degree of customization for the specific DBMS being tested. Specifically, the complexity of the ANSI SQL standard [2, 13] and per-DBMS extensions pose a significant challenge for accurately modeling SQL semantics and building metamorphic relations. For example, consider the handling of NULL values in SQL queries. The behavior of NULL values can be subtle and vary across different DBMSs. In some systems, NULL values are treated as equivalent to an empty string or zero, while in others they are treated as a distinct value that is neither greater than nor less than any other value [1]. Modeling these differences in SQL semantics is a complex task that requires a detailed understanding of the specific features and behaviors of each DBMS.

To address the problem, we present Mozi, a framework that discovers DBMS bugs via configuration-based equivalent transformation as a complement to existing methods. **Instead of focusing on query equivalence, Mozi emphasizes generating equivalent DBMSs with different configurations.** This approach allows Mozi to bypass the need for comprehensive input specification modeling, making it a more efficient and scalable solution for DBMS testing. Mozi finds DBMS bugs with three main steps. First, it executes a query and records the executed plan on the target DBMS. Second, Mozi dynamically modifies specific configurations within the DBMS based on the execution plan on-the-fly, such as optimization strategies, to create a different instantiation of the original version that is equivalent to the query. Finally, it re-issues the same query to the weakened DBMS and compares the results. Designing test oracles based on the comparison can assist in identifying various types of bugs. Discrepancies in the results indicate the presence of correctness bugs (also known as logic bugs). For example, if disabling a join optimization leads to different results, it suggests an error in the optimization's implementation. On the other hand, if a DBMS with some optimizations disabled executes queries faster than the original implementation, it reveals performance bugs, suggesting that these optimizations might be counterproductive. By concentrating on the equivalent transformation, Mozi alleviates the complexity involved in modeling the subtle SQL semantics required by traditional methods.

The main challenge of achieving equivalent transformations is to identify the pertinent configurations and modify them without compromising the query's equivalence. Making arbitrary changes in configurations closely related to the query execution process may result in an inequivalent version of the query, whereas modifying other configurations that are irrelevant to the query execution process may have no effect. For example, suppose a DBMS has a configuration that limits the maximum execution time of queries. If this time limit is shortened arbitrarily, some queries may not produce any results. However, changing other configurations, such as the access control for unrelated tables, may not have any impact on query execution.

We leverage plan-guided transformation to address the challenge. Most modern DBMSs have the ability to generate an execution plan for a given SQL query, which details the steps that the DBMS takes to execute the query [21, 23, 39]. By analyzing the execution plan, it is possible to identify specific configurations (e.g., optimization strategies) that the DBMS is using to execute the query. In turn, this information can be used to change certain configuration parameters to create a modified version of the DBMS that executes the query with different logic but the same results. For example, if the execution plan shows that a specific optimization is being used to speed up the query execution, Mozi can disable this optimization in the modified version of the DBMS. A decrease in response time suggests the possibility of a performance issue within the DBMS. By leveraging the introspection capabilities of the DBMS, Mozi can more easily manipulate and configure the DBMS environment to test different scenarios and configurations.

We conducted extensive experiments to evaluate the effectiveness of Mozi in identifying previously undiscovered bugs on four widely-used DBMSs, namely MySQL [42], MariaDB [34], Clickhouse [9], and PostgreSQL [43]. In the continuous testing of three-weeks, Mozi found a total of 101 previously unknown bugs, including 49 correctness and 52 performance bugs in four DBMSs. Among them, 90 bugs are confirmed and 57 bugs have been fixed. We also evaluate Mozi against some related state-of-the-art testing techniques. In 24-hour experiments, Mozi covers 64,973, 54,464, 43,236, 28,499, and 14,084 more branches, detects 25, 22, 21, 24, and 26 more bugs than PQS [48], NoREC [46], TLP[1] [47], Apollo [25], and Amoeba [33], respectively. Additionally, we assess the significance of the plan-guided approach in Mozi and demonstrate its scalability by applying Mozi to other fuzzers for detecting correctness and performance bugs. In summary, our paper makes the following contributions:

- We present Mozi, a framework for identifying DBMS bugs using configuration-based equivalent transformations, which complements existing methods such as metamorphic testing. Mozi aims to generate different DBMS instantiations that should produce equivalent results when executing the same SQL query. By designing test oracles based on the result comparison, we can detect different types of bugs.
- We propose a three-step process for using Mozi to test DBMSs: first, analyze the execution plan of a generated SQL query; second, manipulate the DBMS configuration to create an equivalent instance; and finally, issue the same SQL query to the new DBMS and compare the results. We offer a solution for efficiently creating a differently configured DBMS by leveraging the analyzed plan of the query to guide the transformation.
- We implement our approach in Mozi and evaluate it in four popular DBMSs against other state-of-the-art techniques. Mozi reported a total of 101 bugs and 90 bugs have been confirmed as previously unknown bugs. It also found more branches and bugs than other techniques.

2 BACKGROUND AND MOTIVATION

DBMS and Configuration. *Database Management Systems* (DBMSs) refer to the software used to manage the storage and retrieval of data in databases [58]. DBMS configuration refers to the settings and parameters that can adjust the behavior of database management systems. These configuration options can affect the performance, reliability, and security of the database system. Table 1 shows common categories of DBMS configurations.

Table 1: Categories of DBMS Configurations

Category	Description
Buffer cache	Size and behavior of the buffer cache.
Concurrency	Control of concurrency and locking.
Memory allocation	Memory allocation and usage.
Disk usage	Control of disk space usage.
Query optimization	Optimization of query execution.
Security	Access control and authentication.

[1]PQS, NoREC, and TLP are three correctness test oracles utilized in SQLancer.

The buffer cache configuration determines the size and behavior of the buffer cache used by the DBMS to store frequently accessed data in memory. The concurrency configuration controls the degree of concurrency and locking used to manage access to shared data. Memory allocation configuration specifies how memory is allocated and used by the DBMS. Disk usage configuration controls how disk space is allocated and used by the DBMS. The optimization configuration refers to the process of optimizing the DBMS performance to ensure fast and efficient data access and processing. It determines how queries are optimized and executed. Examples include the choice of query optimizer, indexing strategies, and parallelism settings. Finally, the security configuration deals with access control and authentication to the database system.

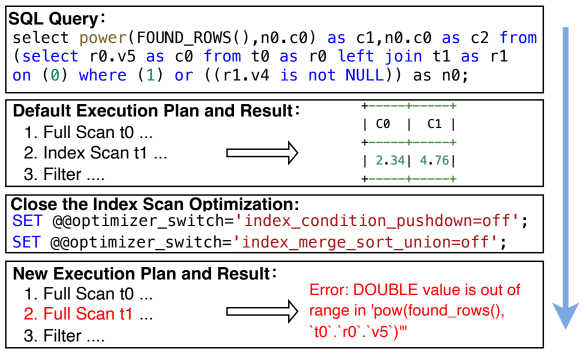
DBMS Metamorphic Testing. *Metamorphic testing* [5, 6, 32, 49] is a common method to construct oracles to detect correctness bugs of DBMSs. A metamorphic relation is a mathematical relationship between the input change and output change of a software system. When the input changes, and the relationship no longer holds for the corresponding output change, it indicates the presence of an error. A common method is like EMI [26] to construct different, but equivalent variants of a query. More concretely, given a query Q and its semantic equivalent query collection C. The collection C can be used to perform a comparison for any DBMS D: if $D(Q) \neq D(Q')$ for some $Q' \in C$, D has a bug. For example, the NoREC [46] method of SQLancer detects the logic bugs of DBMS optimizer by constructing equivalent non-optimization SQL query which can not be optimized by DBMSs based on the existing SQL query. If the non-optimization SQL query gets inconsistent results from the existing SQL query, they detect a logic bug of the optimizer.

However, finding metamorphic relationships for DBMSs is very difficult. Discovering an effective metamorphic relation needs a deep understanding of SQL specification but one relationship can usually focus on only one SQL feature. For instance, the TLP [47] method can only be tested for queries that use `WHERE`, `HAVING`, `GROUP BY`, aggregation functions, and `DISTINCT` features. Besides, the relation may be highly customized for a specific DBMS, which means that adapting it to other DBMSs can be costly.

Basic Idea of Mozi. The key idea of Mozi is to change DBMS itself to an equivalent one as a reference to verify the execution results. Specifically, **rather than finding a mathematical relationship to build a different query, we choose to transform the DBMS into an equivalent one by changing configurations.** Based on the transformation process and query execution results, we can build test oracles to find bugs. For example, when two equivalent DBMSs have different results when executing the same query, there might be a correctness bug in the implementation.

This approach is based on the understanding that changing the configuration can have a critical impact on the system's behavior, often leading to unexpected consequences. For example, enabling or disabling certain query optimizations can significantly affect the performance of the system, while changing the buffer size or cache size can have an impact on the system's memory usage. Moreover, different configurations may cause the DBMS to generate different plans and cover completely different code areas. If the modified configuration can create an equivalent DBMS for the query, then the executed plan should have identical semantics but with different operations. In addition, some configuration settings may interact with each other in non-obvious ways, leading to subtle bugs that are hard to diagnose. Based on these observations, we can define different test oracles to detect various kinds of bugs.

Example: A correctness bug in MySQL. We demonstrate how Mozi can be used to identify bugs in MySQL through an example of a correctness bug. This bug had significant implications for the optimizer's correctness and was confirmed and resolved by MySQL developers. One developer commented, "*It's amazing, I can't imagine that turning off the optimization options would cause such a correctness anomaly*".


SQL Query:

```
select power(FOUND_ROWS(),n0.c0) as c1,n0.c0 as c2 from
(select r0.v5 as c0 from t0 as r0 left join t1 as r1
on (0) where (1) or ((r1.v4 is not NULL)) as n0;
```

Default Execution Plan and Result:

1. Full Scan t0 ...
2. Index Scan t1 ...
3. Filter ....

```
+-----+-----+
| C0  |  C1 |
+-----+-----+
| 2.34| 4.76|
+-----+-----+
```

Close the Index Scan Optimization:

```
SET @@optimizer_switch='index_condition_pushdown=off';
SET @@optimizer_switch='index_merge_sort_union=off';
```

New Execution Plan and Result:

1. Full Scan t0 ...
2. Full Scan t1 ...
3. Filter ....

Error: DOUBLE value is out of range in 'pow(found_rows(), `t0`.`r0`.`v5`)'"

Figure 1: A correctness bug found by Mozi in MySQL. Mozi executes the query and finds the `INDEX SCAN` configuration in the plan. Next, it disables this configuration to instantiate a transformed DBMS. Finally, it re-executes the SQL query and finds inconsistency query results, which indicates a correctness bug.

Figure 1 illustrates how Mozi detected a correctness bug in MySQL. The top part of the figure displays the SQL query that triggers the bug, which was executed with the default configuration by MySQL. The query produced the correct records with the default execution plan, where MySQL scanned the table `t1` using the `INDEX SCAN` method. Mozi then used two `SET` SQL statements to disable the `Index Scan` option of the optimizer and create an equivalent transformed DBMS on-the-fly. Finally, Mozi re-executed the same SQL query in MySQL. However, the query did not execute correctly, and MySQL returned some error messages. As shown in the bottom part of the figure, the new execution plan selected by MySQL used the `FULL SCAN` method to scan table `t1`, resulting in the correctness bug. Detecting the correctness bug we presented using DBMS metamorphic testing is challenging. DBMS metamorphic testing constructs equivalent SQL queries and checks if the results are the same to detect anomalies. However, to trigger this bug, the DBMS configuration must be modified to alter the execution of the same SQL query. As a result, this bug is difficult for other DBMS testing methods to detect within the same time frame.

3 DBMS EQUIVALENT TRANSFORMATION

The main objective of Mozi is to overcome the limitations of traditional approaches that rely on creating different inputs with equivalent semantics by precisely modeling the input specification. Instead, we propose a *DBMS Equivalent Transformation* approach

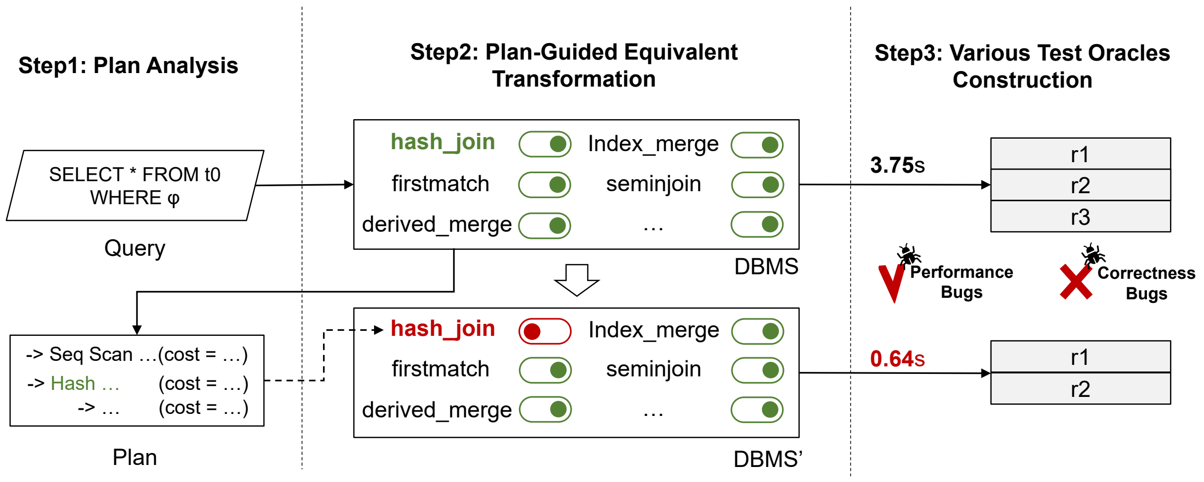


Figure 2: Overview of Mozi. First, Mozi analyzes the plan of a generated query. It sends a generated query into the target DBMS with default configurations. The relevant configurations such as optimization or memory options used in the plan are analyzed. Second, Mozi constructs equivalents of the target DBMS by modifying its configurations in processing the query. Then Mozi re-sends the SQL query to the transformed equivalent DBMS and records the new results. Finally, by comparing the old and new results, Mozi is able to detect performance, correctness bugs, or other types of bugs. For example, different results between the two DBMSs indicate the presence of correctness bugs, while faster execution on the optimization-disabled DBMS indicates the presence of performance bugs.

that generates DBMSs with different configurations for testing purposes. Our idea is based on the premise that equivalent DBMSs with different configurations should produce consistent results when given the same input SQL. We can detect various types of bugs by integrating with various test oracles based on the result comparison. For instance, if there are different results, it indicates the presence of correctness bugs. Moreover, if the execution is faster on the optimization-disabled DBMS, it indicates performance bugs. Figure 2 illustrates the overview of utilizing DBMS equivalent transformation to test bugs with different test oracles. The process consists of three main steps: 1) Plan Analysis, 2) Plan-Guided Equivalent Transformation, 3) Various Test Oracles Construction. The following text in this section will first give some definitions and then present the details of each step.

3.1 Definitions

To better describe the thoughts behind Mozi, we first give some detailed definitions of concepts related to equivalent transformation.

DBMS Configuration Variants. We define $\mathcal{D}$ to be the set of all possible DBMSs, incorporating the various configuration options available. Let C be the set of configurations for the DBMS. We use $Conf(D, c)$ to describe the value for configuration c in DBMS D. DBMS D's configuration variant set D_c is defined as:

$$D_c = \{D' \in \mathcal{D} | \exists c \in C, Conf(D', c) \neq Conf(D, c)\}.$$

DBMS Configuration Transformation. A DBMS Configuration transformation from D to its variant D' ($D' \in D_c$), denoted by $T(D, D')$, is a set of functions that map each different configuration c from D to D', which could be defined as:

$$T(D, D') = \{c1 \rightarrow c2 \in C \times C | c1! = c2\},$$

where $c1 = Conf(D, c)$ and $c2 = Conf(D', c)$.

Query Equivalence. Given a query q and a DBMS D along with its data set d. we use $D(q, d)$ to denote the deterministic result rows retrieved from DBMS D. Given a set of queries Q under some data set d, two DBMS $D, D' \in \mathcal{D}$ are equivalent to Q if and only if

$$\forall q \in Q, D(q, d) = D'(q, d).$$

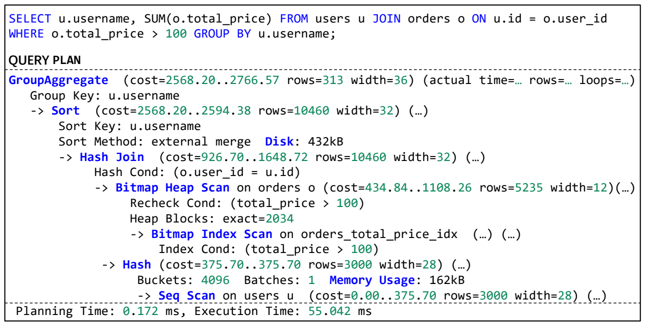
Equivalent Transformation. We use $D_Q = D^e_Q$ to denote that D and D^e are equivalent to the query set Q. Based on the predefined concepts, $T(D, D')$ is a equivalent transformation if and only if

$$D' \in D_c \wedge \{D^e | D_Q = D^e_Q\}.$$

3.2 Plan Analysis

Mozi transforms the configuration based on the query plan. It is built on the understanding that a plan not only reflects the query's execution but also includes the configuration options that influence it. Consider Figure 3 as an example, where we intend to execute a query in PostgreSQL that joins two tables. The figure shows the execution plan that PostgreSQL has devised for this query. It shows that PostgreSQL has chosen to employ a hash join to join the "orders" and "users" tables, as well as a bitmap index scan to filter out orders with a total price greater than 100. The plan also indicates that the "users" table is scanned sequentially.

Using the plan, we can alter the execution process by disabling `enable_hashjoin`, `enable_bitmapscan`, and `enable_seqscan`. Additionally, the plan reveals that certain configuration options were employed to influence it. For instance, the choice of the Hash Join step is determined by the configuration option "work_mem" which determines the memory available for hash joins, sorts, and aggregations. In this instance, the planner estimated that the available



```
SELECT u.username, SUM(o.total_price) FROM users u JOIN orders o ON u.id = o.user_id
WHERE o.total_price > 100 GROUP BY u.username;

QUERY PLAN
-----------------------------------------------------------------------------------
GroupAggregate  (cost=2568.20..2766.57 rows=313 width=36) (actual time=… rows=… loops=…)
   Group Key: u.username
   -> Sort  (cost=2568.20..2594.38 rows=10460 width=32) (…)
       Sort Key: u.username
       Sort Method: external merge  Disk: 432kB
       -> Hash Join  (cost=926.70..1648.72 rows=10460 width=32) (…)
            Hash Cond: (o.user_id = u.id)
            -> Bitmap Heap Scan on orders o (cost=434.84..1108.26 rows=5235 width=12)(…)
                 Recheck Cond: (total_price > 100)
                 Heap Blocks: exact=2034
                 -> Bitmap Index Scan on orders_total_price_idx  (…) (…)
                     Index Cond: (total_price > 100)
             -> Hash (cost=375.70..375.70 rows=3000 width=28) (…)
                  Buckets: 4096  Batches: 1  Memory Usage: 162kB
                  -> Seq Scan on users u  (cost=0.00..375.70 rows=3000 width=28) (…)
-----------------------------------------------------------------------------------
 Planning Time: 0.172 ms, Execution Time: 55.042 ms
```

Figure 3: A plan generated by PostgreSQL that utilizes `Hash Join`, `Index Scan`, and `Seq Scan`.

memory was adequate to execute the hash join efficiently, given the estimated size and selectivity of the input relations. Hence, by adjusting these configurations, we can potentially transform a DBMS into an equivalent one and achieve a different execution process. Therefore, modifying configurations can significantly alter the behavior of a query that has been executed. However, it is challenging to keep query equivalence for transformation.

3.3 Plan-Guided Equivalent Transformation

Ensuring query equivalence between DBMSs while making changes to their configurations that can impact the query execution process based on a test oracle is a challenging task. The changes should be carefully crafted to ensure that meaningful comparisons can be made between the execution results of the original and modified DBMSs with respect to the test oracle while preserving the equivalence. To address the challenge, Mozi proposes the plan-guided configuration transformation approach.

Algorithm 1: Plan-Guided Equivalent Transformation

```
   Input  : Query $Q$, DBMS $D$,
            Original configuration space $C$,
            Test oracle $T$
   Output : Transformed configuration $C'$
1  $P \leftarrow$ extractPlan($Q, D$);
2  $C_P \leftarrow$ getConfigurations($P$);
3  $C_T \leftarrow$ getInfluencedConfigurations($T$);
4  $C' \leftarrow C \setminus (C_P \cap C_T)$ ;
5  foreach $c \in C_P \cap C_T$ do
6      if flipCoin($c$) then
7          $v \leftarrow$ getOnePossibleValue($c$);
8          while $v ==$ getValue($c$) do
9              $v \leftarrow$ getOnePossibleValue($c$);
10         end
11         setValue($c, v$);
12     end
13     $C' = C' \cup c$;
14 end
15 return $C'$;
```

Algorithm 1 shows the overall process of plan-guided equivalent transformation. First, we analyze and extract the plan for the executed query (Line 1). Based on the plan and predefined test oracle, we can identify the candidate configuration sets relevant to both of them (Lines 2-4). In particular, the configurations related to the test oracles will be discussed in Section 3.4. Besides, to ensure equivalence, we only change configurations that are not critical to the query’s correctness but related to the plan. For example, we could change the configurations like memory allocation and disk usage. Some other configurations that are related to low-level functionalities will not be considered, such as “max_relations” which is used to limit the maximum number of tables in the database.

After that, we decided whether to transform the configuration parameter in the candidate sets. For each configuration in the candidate sets, we use the function `flipCoin` to decide whether to change it randomly (Line 6). For each passed configuration, we first calculate a valid value while not equal to the current one (Lines 7-10). If true, then we will set the target value to the configuration and add it to the result set. It is essential to ensure that the target value does not violate any constraints or assumptions and guarantees that the DBMS will still return accurate results. For instance, the amount of memory allocated should not surpass the available memory, and the time limit should be long enough to guarantee the successful retrieval of results.

3.4 Various Test Oracles Construction

Based on the equivalent transformation, we could detect bugs in DBMSs under various test oracles. For example, we could utilize equivalent transformation to conduct correctness testing, performance testing, memory safety testing, authorization testing, and compliance testing. In the following content of this section, we will discuss how to construct test oracles that detect correctness and performance issues. Additionally, we will also explore how to apply equivalent transformations in other test oracles in Section 6.

Test Oracles for Correctness Bugs. Correctness bugs refer to errors in the results of a query. It is also known as logic bugs. To detect correctness bugs, we could directly compare the results from equivalent DBMSs, and check whether the results are different.

One important problem is determining the related configurations to correctness bugs. Specifically, the DBMS configurations need to be modified to guarantee that the execution of queries on two different DBMSs will behave differently but produce consistent results. To achieve this goal, many configurations of certain types can be utilized while testing for correctness errors. First, the optimization configurations can be used to directly alter the query plan and affect query execution. Moreover, some configurations that have indirect impacts could also be considered, such as buffer cache configuration, memory allocation configuration, and disk usage configuration. For instance, a configuration that affects the allocation of memory buffers may not directly impact query processing, but it may lead to memory leaks or corruption, ultimately resulting in incorrect query results or even system crashes.

After the query is re-executed on the transformed DBMS, the results are compared to those obtained from the original DBMS execution. If the values returned by both systems are the same for every query, then the results are considered matched. However, if there are any differences in the returned values, Mozi will flag the results as unmatched and log the differences.

Test Oracles for Performance Bugs. We can also use DBMS equivalent transformation to test performance bugs. A basic way is applying the equivalent transformation process to create a weakened version of the original DBMS, which is designed to have lower performance. If the weakened DBMS significantly performs better than the original DBMS, it indicates a potential performance bug.

In a cost-based optimizer, it is generally expected that disabling optimization operators will not lead to improved database engine performance. If it does, this indicates a performance bug, which can result in slower query processing and reduced system throughput. Such issues can be referred to as performance anomalies caused by reversed optimization. To detect these errors, a DBMS can be weakened by applying configuration transformations such as disabling query optimization or limiting resource usage. The SQL query is then executed on the transformed DBMS, and the plan and execution time are obtained. If the new execution time is lower than the original execution time while the results are equal, a potential performance anomaly is detected. This technique is useful in identifying and resolving performance issues in a cost-based optimizer, which is widely used in popular DBMSs.

For example, in our experiment, the execution time of one SQL query executed by MySQL with the `hash_join` optimization operator was 10.42 seconds. When disabling `hash_join` optimization operators, the new execution time was only 1.54 seconds, which reduces the original execution time by 85%. The decline in performance resulting from optimization implies that this is indeed a performance issue, and the issue has also been confirmed by the developers of MySQL.

4 IMPLEMENTATION

As depicted in Figure 4, the Mozi framework has two layers, namely Transformation Layer and Adaptation Layer.

Transformation Layer. This layer performs DBMS equivalent transformations. The plan analyzer executes SQL commands such as "`EXPLAIN`" [20, 41] to obtain the execution plan and extract the involved SQL operations. The configuration transformer sets configuration on-the-fly by using SQL commands. For example, to disable sequential scan types in the plan derived by PostgreSQL, we can use the SQL command "SET ENABLE_SEQSCAN TO OFF". The result checker compares the results of query executions on equivalent DBMSs based on test oracles.

The test oracles include correctness testing, performance testing, and others. Currently, specific configurations are first classified by category or component related to the oracle and then further determined manually. For example, configurations related to optimizers are often associated with performance issues. Based on that, we have now identified 26 and 21 performance-related configurations for MySQL and PostgreSQL according to their functionalities [19, 38], respectively.

Adaptation Layer. This layer handles adaptations for different target DBMSs with certain specifications. It has a SQL specification analyzer and a query generator. The SQL specification analyzer is used to analyze the SQL specification file to get the supported SQL clauses and configuration lists. The query generator utilizes the analyzed results to produce queries of varying complexity, ranging from straightforward ones to more complex queries that involve joins and subqueries. Specifically, we will extract the BNF paradigm's grammar rules and the clauses supported by the DBMS from the DBMS grammar file. Following this, we will engage in SQL statement modeling. During the query generation process, the generator first constructs the basic skeleton of the syntax tree. Subsequently, it recursively selects clauses to populate nodes with a certain probability, generating queries that encompass a broader range of configurations. This process establishes a comprehensive testing ground for various scenarios and optimizations. In addition, the generator avoids generating queries with non-deterministic behaviors. These behaviors may confuse oracles and lead to false alarms. For example, the generator avoids using functions like random() or ever-changing environment variables like time.

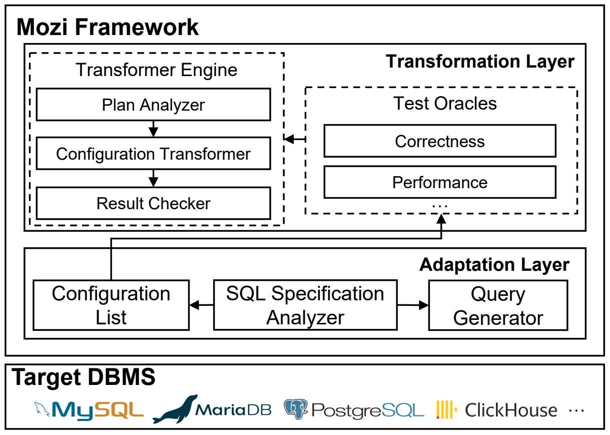


Figure 4: The core components of Mozi.

5 EVALUATION

In this section, we evaluate the effectiveness of detecting bugs with the test oracle constructed by Mozi. Our evaluation aims to answer the following questions:

- **RQ1:** Can Mozi find previously unknown correctness and performance bugs?
- **RQ2:** How does Mozi perform compared to other DBMS testing techniques?
- **RQ3:** How effective is the plan-guided transformation for Mozi in detecting the bugs?
- **RQ4:** Can Mozi help other fuzzers find bugs?

5.1 Evaluation Setup

We evaluated Mozi on four widely used open-source DBMSs, namely MySQL (8.0.32), MariaDB (10.8), Clickhouse (22.11.4.3), and PostgreSQL (15.2). The experiments were conducted 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. Each docker container is allocated with 5 CPU cores and 40 GiB of RAM.

5.2 DBMS Vulnerability Detection

Bug Statistics. We applied Mozi to test performance and correctness bugs in three weeks. The four evaluated DBMSs are widely used and well-tested. Nevertheless, Mozi performed well and finally found 101 previously unknown bugs in them. We also used SQLancer [45] (with test oracles PQS [48], TLP [47], and NoREC [46]), Apollo [25], and Amoeba [33] to test these DBMSs, but they can only find a subset of these bugs (shown in Section 5.3).

Table 2 shows the statistics of the bugs. Mozi reported a total of 49 correctness bugs and 52 performance bugs. We reported all the bugs to the corresponding DBMS developers. Among them, 46 correctness bugs and 44 performance bugs have been confirmed as previously unknown bugs. At the time of writing this paper, 57 bugs have been fixed, and we have received gratitude from the developers. In addition, 11 bugs are still being validated by the developers due to the complexity of DBMS.

Table 2: Number of reported and confirmed bugs by Mozi in three weeks.

	Correctness		Performance	
DBMS	Reported	Confirmed	Reported	Confirmed
MySQL	14	13	21	19
MariaDB	14	14	21	17
Clickhouse	12	11	8	6
PostgreSQL	9	8	2	2
Total	49	46	52	44

Bug Severity. Based on the analysis of the DBMS developer, the bugs detected by Mozi are *distributed in over 35 components on the tested DBMSs.* The correctness bugs are hard to detect because they will not cause obvious signs such as system crashes. The performance bugs are also critical because they could influence the overall response time of the DBMS. More importantly, during our communication with the developers, they expressed their interest in discovering vulnerabilities by constructing an equivalent DBMS. And they discovered many issues that they had never focused on in the development. At the time of writing this paper, developers have fixed a total of 57 bugs.

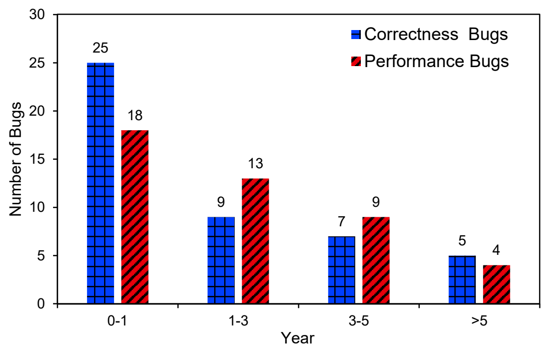


Figure 5: Time distribution of imported codes for the confirmed bugs. Among them, 25 bugs had remained latent for over 3 years, while 43 were imported within the last 1 year.

Moreover, many bugs had been present in the DBMSs for many years before being detected. Figure 5 displays the time distribution of the imported error codes for the confirmed bugs. As shown in the figure, 25 bugs had remained latent for over 3 years, while 43 bugs were imported within the last 1 year. Without Mozi, these bugs may remain for more time and result in potential harm. Here we present a performance bug in PostgreSQL only found by Mozi.

Case Study. Figure 6 illustrates the process of triggering a significant performance bug in PostgreSQL. The bug causes a considerable decrease in PostgreSQL's performance on the `index_scan` optimization, a classical technique for optimizing SQL queries by efficiently accessing data through indexes. Normally, using `index_scan` should result in faster data retrieval than performing a full table scan using `full_scan`. However, utilizing the `index_scan` optimization in this instance led to a 20x decrease in performance compared to not using it.

```
-- Step 1. Create the init tables
create table t1 ( v0 int, v1 varchar ,... ) ;
create table t2 ( v0 CURRENT_TIMESTAMP,v1 float,... ) ;
create table t3 ( v0 float primary key,v1 float,... ) ;
-- Step 2. Add index to the tables
create index i1 on t1 (v1) using hash;
...
-- Step 3. Insert records to tables
insert into v1 value(1,'v5easaw',...);
...
-- Step 4. The SQL query for execution: Execution Time:11.42s
select case when (EXISTS (select ref1.v as c0, ...
from table1 as ref1 where 1)) and (s0.c0 is not NULL)
then s0.c0 else s0.c0 end as c0, MONTH(s0.c1) as c1,
JSON_OBJECT() as c2, case when s1.c0 is not
NULL then s1.c0 else s1.c0 end (select ...;
-- Step 5. Close the index_scan optimization
set enable_indexscan="off"
-- Step 6. Execute the same SQL query, Execution Time: 0.534s
```

Figure 6: The simplified PoC to trigger the performance bug in PostgreSQL. The bug is influenced by the `index_scan` optimization, which was only detected by Mozi.

The mechanism to trigger the bug. Figure 6 illustrates the steps to trigger bugs in PostgreSQL. In Step ①, Mozi creates complex initial tables. It then adds constraints, such as indexes, to the tables in Step ②, followed by inserting a large number of records into them in Step ③. In Step ④, Mozi generates a SQL query that includes some complex clauses and requires scanning some records from table `t1`, which triggers the `index_scan` optimization of PostgreSQL. As shown, PostgreSQL takes 11.42 seconds to execute the SQL query. However, after disabling the `index_scan` optimization in Step ⑤, PostgreSQL only takes 0.534 seconds to execute the same SQL query. The performance decrease caused by using the `index_scan` optimization is significant, leading to a 20x slowdown compared to not using the optimization in execution.

Why was the bug only discovered by Mozi? The bug involves DBMS configuration changes, which can hardly be detected by other DBMS testing methods. Specifically, SQLancer and Amoeba detect the bugs by constructing the equivalent SQL query, which does not change the configuration of DBMS. Therefore, this bug can not be discovered by them. Similarly, Apollo detects the regression performance bugs of DBMSs by comparing the response time of two different-version DBMSs with the same configuration, thus it cannot detect this kind of bug. In contrast, Mozi detected this bug by constructing a test oracle for performance bug detection with configuration-based equivalent transformation. In summary, the results indicate that Mozi can detect previously unknown correctness and performance bugs, which adequately answers **RQ1**.

5.3 Comparison with Other Techniques

We implement Mozi with correctness and performance test oracles into Mozi ^{cor} and Mozi ^{per} to detect correctness bugs and performance bugs, respectively.

Compared Techniques and Setup. To evaluate the effectiveness of our framework, Mozi, we conducted a comparison study between Mozi ^{cor} and Mozi ^{per} with the state-of-the-art DBMS testing methods. Specifically, we compared Mozi ^{cor} against SQLancer using three logic test oracles, namely PQS [48], NoREC [46], and TLP [47] for correctness bug detection. In the following text, we will refer to these three implementations of SQLancer by their respective oracle names. We also compared Mozi ^{per} with Apollo [25] and Amoeba [33] for performance bug detection. Currently, Amoeba only supports PostgreSQL. We ran the testing tools on each DBMS for 24 hours and recorded the number of triggered bugs. All bugs were reported to the developers, and we used the confirmed number of bugs as the result. To ensure a fair comparison, we collected the generated queries from each testing method and dry-ran the queries to ensure uniform branch coverage.

Results. Mozi outperforms other testing methods in detecting both performance and correctness bugs. Table 3 displays the number of bugs detected by each method. In 24 hours, Mozi ^{cor} detected a total of 29 bugs. Compared to PQS, NoREC, and TLP, Mozi ^{cor} found 25, 22, and 21 more bugs, respectively. For performance bugs, Mozi ^{per} discovered 27 bugs, while Apollo and Amoeba detected 3 and 1 bugs on the tested DBMSs, respectively.

We investigated the bugs and found that the bugs identified by Mozi did not overlap with other techniques. This is because Mozi detects problems through the transformation of DBMS configurations, which is a process not employed by the other methods. On the other hand, Mozi did not find the bugs detected by other techniques. This disparity arises because PQS, NoREC, and TLP employ highly customized test oracles, with the latter two incorporating statement changes not utilized in Mozi ^{cor}. The test oracles of Amoeba and Apollo for identifying performance bugs differ from Mozi ^{per}. The first one necessitates constructing an equivalent statement, while the other requires referencing the previous version. In contrast, Mozi utilizes DBMS transformation, resulting in a low likelihood of identifying common bugs.

Table 3: The number of detected bugs by each DBMS testing method in 24 hours.

	Correctness				Performance		
DBMS	PQS	NoREC	TLP	Mozi ^{cor}	Apollo	Amoeba	Mozi ^{per}
MySQL	1	2	2	9	1	–	11
MariaDB	2	3	2	10	0	–	13
Clickhouse	0	1	3	6	1	–	2
PostgreSQL	1	1	1	4	1	1	1
Total	4	7	8	29	3	1	27
Improvement	25↑	22↑	21↑	-	24↑	26↑	-

Table 4 shows the number of branches covered by each method in 24 hours. It shows that Mozi covers more branches in 24 hours when compared to other testing methods. Specifically, compared to three correctness bug testing methods, Mozi ^{cor} covered a total of 64,973, 54,464, and 43,236 more branches than PQS, NoREC, and TLP, respectively. And compared to the performance bug testing fuzzer Apollo, Mozi ^{per} covered a total of 28,499 more branches in MySQL, MariaDB, Clickhouse, and PostgreSQL. Compared to Amoeba, Mozi covered 14,084 more branches in PostgreSQL.

Table 4: The number of covered branches by each DBMS testing method in 24 hours.

	Correctness				Performance		
DBMS	PQS	NoREC	TLP	Mozi ^{cor}	Apollo	Amoeba	Mozi ^{per}
MySQL	50,294	52,984	60,274	75,293	70,923	–	74,836
MariaDB	40,154	42,938	45,792	54,982	46,365	–	55,387
Clickhouse	65,980	69,842	69,928	79,852	72,388	–	80,142
PostgreSQL	59,811	60,984	61,982	71,085	63,748	56,844	70,928
Total	216,239	226,748	237,976	281,212	252,794	56,844	281,293
Improvement	64,973↑	54,464↑	43,236↑	-	28,499↑	14,084↑ [*]	-

[*] The improvement is only for PostgreSQL since Amoeba only supports it among four DBMSs at the time of writing.

The main reason for Mozi's improvement in bug detection and branch coverage is its independence from specific SQL features in test oracle. This flexibility enables Mozi to test a broader range of functionalities within the DBMS. Specifically, PQS, NoREC, TLP, and Amoeba are constructed based on specific features to create test cases, which limits the SQL grammar they can support. For example, NoREC detects logic bugs in the DBMS optimizer by constructing equivalent optimized and unoptimized queries based on the optimizer rules. However, this approach only covers SQL grammar that conforms to the optimizer rules. In contrast, Mozi generates SQL queries without any limitations on the SQL grammar since it relies on the DBMS itself to evaluate query results. As a result, Mozi can support more SQL grammar and cover more branches compared to PQS, NoREC, and TLP, which explains why it can detect more correctness bugs in these DBMSs.

Apollo generates queries by comparing the query performance of two versions of the same DBMS, which is effective for detecting performance regression bugs. However, Apollo is limited by utilizing differential testing on current and previous DBMS versions. The bugs that exist in both versions will not be detected. In addition, this approach relies heavily on the differences between the two versions, which can limit the diversity of its generated queries. If the differences between versions are minimal, Apollo may not be able to generate a sufficient number of unique queries to fully test the different logic in the DBMS. Differently, Mozi transforms the DBMS based on the executed query. It expands the differences by changing configurations thus more DBMS behaviors could be triggered. Moreover, in cases where there are only a limited number of versions of DBMSs available, Apollo may encounter difficulty in applying. The result illustrates that Mozi can test more functionality of the DBMS and cover more branches, leading to the detection of more performance bugs, which adequately answers **RQ2**.

5.4 Importance of Plan-Guided Algorithm

We implement Mozi $^{!\rho}$ to measure the importance of the plan-guide transformation in Mozi. It disables the plan-guided component, which randomly changes the configuration without any guidance from the execution plan of the SQL query. Note that there are two versions of Mozi $^{!\rho}$: Mozi $^{!\rho cor}$, which uses the correctness test oracle, and Mozi $^{!\rho per}$, which uses the performance test oracle. We run Mozi ^{cor} and Mozi ^{per} against their plan-guided disabled versions for 24 hours and collect the number of detected bugs and covered branches for comparison.

Table 5: Number of detected correctness bugs and covered branches of Mozi ^{cor} and Mozi $^{!\rho cor}$ in 24 hours.

DBMS	Bugs Number		Branch Coverage	
Name	Mozi ^{cor}	Mozi $^{!\rho cor}$	Mozi ^{cor}	Mozi $^{!\rho cor}$
MySQL	9	5	75,293	73,428
MariaDB	10	3	54,982	49,872
Clickhouse	6	1	79,852	73,293
PostgreSQL	4	2	71,085	62,396
Total	29	11	281,212	258,989

Table 5 shows the number of detected correctness bugs and covered branches by Mozi ^{cor} and Mozi $^{!\rho cor}$. The table illustrates that the plan-guided algorithm helps Mozi find more correctness bugs and cover more branches. Specifically, Mozi ^{cor} triggered 4, 7, 5, and 2 more correctness bugs and covered 1865, 5110, 6559, and 8689 more branches when compared to Mozi $^{!\rho cor}$ on MySQL, MariaDB, Clickhouse, and PostgreSQL, respectively.

Table 6: Number of detected performance bugs and covered branches of Mozi ^{per} and Mozi $^{!\rho per}$ in 24 hours.

DBMS	Bugs Number		Branch Coverage	
Name	Mozi ^{per}	Mozi $^{!\rho per}$	Mozi ^{per}	Mozi $^{!\rho per}$
MySQL	11	4	74,836	70,825
MariaDB	13	5	55,387	48,984
Clickhouse	2	0	80,142	73,475
PostgreSQL	1	0	70,928	64,871
Total	27	9	281,293	258,155

Table 6 shows the number of detected performance bugs and covered branches by Mozi ^{per} and Mozi $^{!\rho per}$. First, Mozi ^{per} found more performance bugs than the plan-guided disabled version. Specifically, Mozi found 7, 8, 2, and 1 more performance bugs when compared to Mozi $^{!\rho per}$ on MySQL, MariaDB, Clickhouse and PostgreSQL, respectively. Furthermore, it is possible that more performance bugs arose from higher branch coverage. More concretely, in the testing, Mozi ^{per} covered 4011, 6403, 6667, and 6057 more branches compared to Mozi $^{!\rho per}$ on MySQL, MariaDB, Clickhouse, and PostgreSQL, respectively.

From the results, we can see that the plan guidance plays an important role in the performance of Mozi. It not only helps Mozi cover more states in target DBMSs and also helps to find more bugs. The main reason is that plan guidance helps Mozi find the configurations related to the execution processes of the SQL query. By changing these configurations, the optimization plan might be changed significantly. Thus many new behaviors will be triggered and new branches will be covered. When disabling the plan guidance, Mozi $^{!\rho}$ transforms the DBMS by arbitrarily modifying the configuration. However, arbitrarily changing might not influence the covered logic of the specific query. Consequently, Mozi could cover more branches than Mozi $^{!\rho}$. Therefore, plan guidance plays a crucial role in effectively covering branches and successfully detecting bugs, which adequately answers **RQ3**.

5.5 Scalability of Mozi to Other Fuzzers

SQLsmith and Squirrel are two widely used fuzzers targeting crash bugs. To measure the scalability of utilizing Mozi to help other tools find bugs, we adapt the two test oracles of Mozi to these two fuzzers to detect performance bugs and correctness bugs in DBMS. Specifically, we implement SQLsmith+Mozi ^{cor} and Squirrel+Mozi ^{cor} to detect correctness bugs. We also implement SQLsmith+Mozi ^{per} and Squirrel+Mozi ^{per} to detect the performance bugs. Note that we only use 294 and 378 lines of C++ codes to adapt the two test oracles to SQLsmith, 287 and 391 lines of C++ codes for Squirrel, respectively. The test oracles in Mozi are constructed using a configuration-based equivalent transformation approach that is independent of SQL generation. Therefore, Mozi can easily adapt to other SQL generators and perform well in detecting bugs. Specifically, for every query generated by them, we initially execute it using the original configuration of the target DBMS. Subsequently, guided by the plan, we transform the configuration and re-run the query to detect bugs. After that, the original configuration is restored, and the tools are employed to generate the next query. We run these tools on the four test DBMSs for 24 hours and compare them with SQLsmith and Squirrel, respectively. Note that SQLsmith+Mozi ^{cor}, SQLsmith+Mozi ^{per}, Squirrel+Mozi ^{cor}, and Squirrel+Mozi ^{per} can still detect crash bugs because we do not close the basic functionality of each tool.

Table 7 shows the number of crash bugs, correctness bugs, and performance bugs detected by each tool in 24 hours. It shows that Mozi performs well when scaled to other fuzzers. First, Mozi helps SQLsmith and Squirrel detect correctness bugs and performance bugs with the two test oracles. Specifically, SQLsmith+Mozi ^{cor} and Squirrel+Mozi ^{cor} discovered 13 and 7 correctness bugs apart from crash bugs, respectively. Similarly, SQLsmith+Mozi ^{per} and Squirrel+Mozi ^{per} also detected 14 and 12 performance bugs in addition to the crash bugs, respectively. By combining the test oracle built by equivalent transformation, these tools are given the ability to find correctness and performance issues. Based on the generated SQL queries, Mozi transforms the target DBMSs into equivalent ones, resulting in potentially different behaviors. This allows for the comparison of query results, enabling the detection of correctness and performance bugs.

Moreover, Mozi does not significantly diminish the effectiveness of SQLsmith and Squirrel in detecting crash bugs. Specifically, SQLsmith+Mozi ^{cor} and SQLsmith+Mozi ^{per} only missed one crash bug each when compared to SQLsmith. This is because

Table 7: Number of detected bugs for Mozi augmented SQLsmith and Squirrel in 24 hours. Note that SQLsmith+Mozi ^{cor} and Squirrel+Mozi ^{cor} are the versions augmented to test correctness bugs, while SQLsmith+Mozi ^{per} and Squirrel+Mozi ^{per} are the versions augmented to test performance bugs.

	SQLsmith	SQLsmith+Mozi ^{cor}		SQLsmith+Mozi ^{per}		Squirrel	Squirrel+Mozi ^{cor}		Squirrel+Mozi ^{per}	
DBMS	Crash	Crash	Correctness	Crash	Performance	Crash	Crash	Correctness	Crash	Performance
MySQL	2	2	2	2	5	1	1	2	2	4
MariaDB	3	2	5	3	4	2	2	2	2	6
Clickhouse	1	1	2	1	2	1	1	1	1	1
PostgreSQL	4	4	4	3	3	2	2	2	2	1
Total	10	9	13	9	14	6	6	7	7	12

Mozi requires multiple executions of each SQL query and additional SQL statements to alter the DBMS configuration. In other words, the augmented versions executed fewer SQL queries and thus missed one bug. However, modifying the DBMS configuration and re-executing SQL queries can potentially trigger new behaviors in the DBMS, which can aid in identifying new bugs. For example, Squirrel+Mozi ^{cor} detected the same 6 crash bugs as Squirrel, and Squirrel+Mozi ^{per} identified one additional crash bug compared to Squirrel. In summary, Mozi can help other fuzzers find more bugs of various types, which adequately answers **RQ4**.

6 DISCUSSION

Adaptability of Mozi. Mozi is adaptable to DBMSs which support changing configurations during runtime. For a new DBMS, Mozi will automatically derive the related SQL commands and configurations from SQL specifications. The only human cost involved is deciding whether these configurations are relevant to the specific oracle. The cost is controllable because the specific test oracle is related to certain types and the number of one configuration type is generally not large. For example, there are a total of 340 configurations for PostgreSQL and 21 for optimizations related to performance bugs [18].

Alternative Transformation Methods. Besides changing configurations, other methods also exist for transforming DBMSs. For example, we could generate various DBMS versions by compiling the source code using different compilers or optimization levels [22], ensuring that these versions are equivalent. Another approach involves detecting the code covered by a given query, removing any unrelated code, and recompiling to generate a query-equivalent DBMS. However, these methods require recompiling the entire DBMS, which can be time-consuming, especially for large and complex DBMSs. An alternative approach is to use on-the-fly recompiling techniques [56]. However, it still imposes a substantial overhead, making it challenging to adapt to DBMS transformations.

Other Potential Test Oracles. DBMS equivalent transformation can be used for testing various types of bugs with other test oracles, in addition to correctness and performance bugs. For example, *memory safety testing* can be performed by creating DBMSs with varied memory configurations and generating test cases to exercise memory allocation, deallocation, and accesses. The process can be combined with dynamic analysis tools such as AddressSanitizer [51]. Moreover, *Authorization testing* can be performed by creating equivalent DBMSs with different security configurations to verify access controls and permissions. By generating alternative DBMSs with varying security settings, we can thoroughly examine whether certain accesses violate the intended restrictions.

7 RELATED WORK

In this section, we will first introduce DBMS Fuzzing, and then focus on works related to DBMS test oracle construction and configuration settings.

DBMS Fuzzing. Fuzzing [7, 29, 31, 37, 60] has been applied to DBMSs in recent years and has found hundreds of bugs. These DBMS fuzzers generate SQL test cases continuously, feed them to the target DBMS, and check whether the DBMS works normally when executing them. Most fuzzers [3, 16, 24, 28, 30, 50, 57, 59, 62] focus on generating complex and valid SQL test cases to find the memory safety bugs. They could be roughly divided into generation-based and mutation-based fuzzers. Generation-based fuzzers generate SQL queries based on predefined SQL generation models. SQLsmith [50] generates SELECT statements with complex structures according to its predefined AST models. Amoeba [33] leverages domain-specific design languages to generate SQL queries from scratch based on the schema of the database. Mutation-based fuzzers mutate existing SQL queries to produce new queries, which are always guided by coverage information. Squirrel [62] designs an intermediate representation (IR) to mutate SQL test cases with code coverage guidance. DynSQL [24] establishes a mapping model between binary files and SQL query syntax trees. Lego [28] generates SQL statement sequences with abundant types by exploring SQL statements of different types and analyzing type affinities. Griffin [16] proposes metadata graphs of SQL statements to provide a grammar-free way for mutation of SQL test cases. Unicorn [59] utilizes hybrid input synthesis to generate queries with time-series elements. QPG [3] in SQLancer introduces a mutation technique to use DDL (e.g., CREATE) and DML (e.g., INSERT) statements to change the database state. It saves those mutated cases that cover unseen query plans to explore more behaviors. SQLRight [30] separates SELECT and other statements for mutation, and finally concatenates them together.

Unlike these works that generate complex SQL queries to test database management systems, Mozi takes a different approach. Instead, it is a framework that applies equivalent transformations to DBMSs. By using this technique, Mozi is able to test both correctness and performance bugs. Nevertheless, The SQL generation technique and DBMS equivalent transformation are orthogonal. As a result, it is possible to adapt the SQL generation technique into Mozi to combine the advantages of both techniques.

Metamorphic Testing. Many works [30, 33, 46–48, 53] detect errors in a DBMS by verifying whether the execution results of SQL statements conform to predefined rules. Amoeba [33] aims to find performance bugs in DBMS by constructing semantically equivalent query pairs and comparing the response times when the DBMS executes them. NOREC [46] and TLP [47] are two test oracles used in SQLancer. NoREC converts a SQL query to one that cannot be optimized by optimizers and compares the two's execution results to find optimizer bugs. TLP employs the concept of partitioning, wherein a problem is identified by partitioning the original query into multiple more complex queries. The approach involves comparing the results of the original query with the composed result of the partitioned queries to determine their consistency. DQE [53] constructs SQL statements that should access the same rows and verifies whether the rows fetched by these statements are the same during execution. The EMI (Equivalence Modulo Inputs) [26] can be seen as a special case of metamorphic testing. It aims to reduce the complexity of validating the correctness of compilers. It mainly focuses on generating the EMI variants (i.e., the equivalent programs under some test inputs) by randomly deleting or keeping the unexecuted statements on the input set.

Compared to metamorphic testing, our work focuses specifically on changing the DBMS itself. While metamorphic testing relies on manually defining metamorphic relations, our approach automatically generates equivalent DBMSs that can be used for a variety of testing purposes, including performance and correctness testing. Some other test oracles could also be extended like authorization testing and compliance testing.

Differential Testing. Some approaches [12, 25, 52] are proposed to test the same SQL statements on different DBMSs, known as differential testing [35]. RAGS [52] proposes the idea of validating the output of SQL statements by comparing the execution results of multiple DBMS vendors. Apollo [25] compares the executing speed of the same SQL statements on different versions of the target DBMS to detect its performance bugs. DT2 [12] finds transaction discrepancies on multiple MySQL-compatible DBMSs by comparing their transaction execution results of the same transactions.

Mozi utilizes a special form of *differential testing* to test correctness and performance bugs. Compared to other works that use one fixed DBMS for comparison, Mozi continuously transforms the configuration of a DBMS to compare it dynamically with different instantiations. Due to the equivalence with respect to queries, Mozi circumvents the problem of not being able to find a DBMS for comparison or only being able to test limited queries with a common syntax in both compared DBMSs.

Configuration Tuning. Configuration tuning techniques [14, 27, 54, 55, 61] are already used in DBMS for performance analysis and improvement. To recommend improved configurations, iTuned [14] automatically finds possible high-performance parameter settings and verifies them by running online experiments in production database environments. Many methods use machine learning methods to achieve automated tuning [27, 54, 55]. For example, OtterTune [54] employs large-scale machine learning to automate DBMS tuning by analyzing past experience and new execution information. QTune [27] is a query-aware database tuning system that uses a deep reinforcement learning model to efficiently and effectively tune database configurations.

These works can improve the DBMS performance by configuration tuning, but they ignore the possible bugs under different configurations. Mozi can find such bugs following the DBMS equivalent transformation.

8 CONCLUSION

In this paper, we proposed Mozi, a framework for discovering bugs in DBMSs via configuration-based equivalent construction. The framework involves issuing a query to the DBMS, changing configurations on-the-fly, and re-issuing the query to the modified DBMS to compare results. Different results indicate correctness bugs, while faster execution on the weakened DBMS indicates performance bugs. Our experiment results show that Mozi detected many correctness and performance bugs that were missed by other tools, indicating its superiority in terms of bug-finding capabilities. In our future work, we will focus on studying other transformation methods and finding other potential test oracles.

ACKNOWLEDGMENTS

This research is sponsored in part by the NSFC Program (No. 62302256, 62022046, 92167101, 62021002), Chinese Postdoctoral Science Foundation(2023M731953), and National Key Research and Development Project (No. 2022YFB3104000).

REFERENCES

[1] 2023. Null Value Differences in RDBMS. https://www.linkedin.com/pulse/null-value-differences-rdbms-sumit-sengupta. Accessed: January 18, 2024.

[2] 2024. SQL Language Reference–ANSI Standards. https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/ANSI-Standards.html#GUID-F51EA195-0669-4DED-9D81-B7205AAC642F/. Accessed: January 18, 2024.

[3] Jinsheng Ba and Manuel Rigger. 2023. Testing database engines via query plan guidance. In *Proceedings of International Conference on Software Engineering (ICSE)*.

[4] Adam Bannister. 2021. 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. Accessed: January 18, 2024.

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

[6] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–27.

[7] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *USENIX Security Symposium*.

[8] Catalin Cimpanu. 2019. Google Chrome impacted by new Magellan 2.0 vulnerabilities. https://www.zdnet.com/article/google-chrome-impacted-by-new-magellan-2-0-vulnerabilities/. Accessed: January 18, 2024.

[9] ClickHouse 2024. ClickHouse Website. https://clickhouse.com/. Accessed: January 18, 2024.

[10] PostgreSQL Community. 2014. PostgreSQL Bug List. https://www.postgresql.org/list/pgsql-bugs/. Accessed: January 18, 2024.

[11] Carlos Coronel and Steven Morris. 2019. *Database systems: design, implementation and management.* Cengage learning.

[12] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. 2022. Differentially Testing Database Transactions for Fun and Profit. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[13] Chris J Date. 1989. *A Guide to the SQL Standard.* Addison-Wesley Longman Publishing Co., Inc.

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

[15] R Elmasri, SB Navathe, R Elmasri, and SB Navathe. 2015. Fundamentals of Database Systems. In *Advances in Databases and Information Systems*. Springer, 139.

[16] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-free DBMS fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[17] James R Groff, Paul N Weinberg, and Andrew J Oppel. 2002. *SQL: the complete reference*. Vol. 2. McGraw-Hill/Osborne.

[18] The PostgreSQL Global Development Group. 2023. PostgreSQL 14.10 Documentation. https://www.postgresql.org/files/documentation/pdf/14/postgresql-14-A4.pdf. Accessed: January 18, 2024.

[19] The PostgreSQL Global Development Group. 2023. Query Planning. https://www.postgresql.org/docs/current/runtime-config-query.html. Accessed: January 18, 2024.

[20] The PostgreSQL Global Development Group. 2024. EXPLAIN Statement of PostgreSQL. https://www.postgresql.org/docs/current/sql-explain.html. Accessed: January 18, 2024.

[21] The PostgreSQL Global Development Group. 2024. Using EXPLAIN. https://www.postgresql.org/docs/current/using-explain.html. Accessed: January 18, 2024.

[22] Kenneth Hoste and Lieven Eeckhout. 2008. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. 165–174.

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

[24] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation.

[25] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2020. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*. Tokyo, Japan.

[26] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.

[27] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.

[28] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-oriented DBMS fuzzing. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*.

[29] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. PAFL: Extend Fuzzing Optimizations of Single Mode to Industrial Parallel Mode.

[30] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of {DBMS} with Coverage-based Guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. 4309–4326.

[31] LibFuzzer 2024. LibFuzzer. https://www.llvm.org/docs/LibFuzzer.html. Accessed: January 18, 2024.

[32] Huai Liu, Fei-Ching Kuo, Dave Towey, and Tsong Yueh Chen. 2013. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering* 40, 1 (2013), 4–22.

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

[34] MariaDB 2024. MariaDB. https://mariadb.org/. Accessed: January 18, 2024.

[35] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[36] Jim Melton. 1996. Sql language summary. *Acm Computing Surveys (CSUR)* 28, 1 (1996), 141–143.

[37] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990).

[38] MySQL 2014. MySQL 8.0 Reference Manual, Switchable Optimizations. https://dev.mysql.com/doc/refman/8.0/en/switchable-optimizations.html. Accessed: January 18, 2024.

[39] MySQL 2024. Understanding the Query Execution Plan. https://dev.mysql.com/doc/refman/8.0/en/execution-plan-information.html. Accessed: January 18, 2024.

[40] Oracle. 2014. MySQL Bug List. https://bugs.mysql.com/. Accessed: January 18, 2024.

[41] Oracle. 2024. EXPLAIN Statement of MySQL. https://dev.mysql.com/doc/refman/8.0/en/explain.html. Accessed: January 18, 2024.

[42] Oracle 2024. MySQL. https://www.mysql.com/. Accessed: January 18, 2024.

[43] PostgreSQL 2024. PostgreSQL. https://www.postgresql.org/. Accessed: January 18, 2024.

[44] Manuel Rigger. 2024. Bugs found in Database Management Systems. https://www.manuelrigger.at/dbms-bugs. Accessed: January 18, 2024.

[45] Manuel Rigger. 2024. SQLancer Website. https://github.com/sqlancer/sqlancer. Accessed: January 18, 2024.

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

[47] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[48] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation OSDI 20)*. 667–682.

[49] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on software engineering* 42, 9 (2016), 805–824.

[50] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. 2018. SQLsmith: a random SQL query generator. https://github.com/anse1/sqlsmith

[51] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[52] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, New York, USA*. Morgan Kaufmann, 618–622.

[53] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing Database Systems via Differential Query Execution. In *Proceedings of International Conference on Software Engineering (ICSE)*.

[54] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.

[55] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien, and Andrew Pavlo. 2021. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1241–1253.

[56] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. 2022. Odin: on-demand instrumentation with on-the-fly recompilation. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1010–1024.

[57] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 328–337. https://doi.org/10.1109/ICSE-SEIP52600.2021.00042

[58] Wikipedia 2024. databases. https://en.wikipedia.org/wiki/Database. Accessed: January 18, 2024.

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

[60] Michał Zalewski. 2017. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: January 18, 2024.

[61] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards dynamic and safe configuration tuning for cloud databases. In *Proceedings of the 2022 International Conference on Management of Data*. 631–645.

[62] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing Database Management Systems with Language Validity and Coverage Feedback. In *The ACM Conference on Computer and Communications Security (CCS), 2020*.