Computers & Security 157 (2025) 104564

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Full length article

Detecting DBMS bugs with context-sensitive instantiation and multi-plan execution

Jiaqi Li [a,b], Ke Wang [a], Yaoguang Chen [b], Yajin Zhou [a,*], Lei Wu [a], Jiashui Wang [a,b]

[a] College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang, China
[b] Department of Security Countermeasure Technology, Ant Group CO Ltd, Hangzhou, Zhejiang, China

ARTICLE INFO

Dataset link: https://github.com/anonymous44117/Kangaroo

Keywords:
Database security
Software testing
Fuzzing

ABSTRACT

DBMS (Database Management System) bugs can cause serious consequences, posing severe security and privacy concerns. This paper works towards the detection of crash-related bugs and logic bugs in DBMSs, and aims at solving the two innate challenges, including how to generate semantically correct SQL queries in a test case, and how to propose effective oracles to capture logic bugs. To this end, our system proposes two key techniques. The first key technique is called context-sensitive instantiation, which can obtain all static semantic requirements to guide query generation. The second key technique is called multi-plan execution, which can effectively capture logic bugs. Given a test case, multi-plan execution makes the DBMS execute all query plans instead of the default optimal one, and compares the results. A logic bug is detected if a difference is found among the execution results of the executed query plans. We have implemented a prototype system called Kangaroo and applied it to three widely used and well-tested DBMSs, including SQLite, PostgreSQL, and MySQL. Our system successfully detected 54 previously unknown bugs, including 41 crash-related bugs and 13 logic bugs. The comparison between our system with the state-of-the-art systems shows that our system outperforms them in terms of the number of generated semantically valid SQL queries, the explored code paths during testing, and the detected bugs.

1. Introduction

Database management systems (DBMSs) provide fundamental infrastructure for many applications (Anon, 2022k,f), so it is crucial that they can be relied upon. DBMS bugs could result in data leakage, data manipulation, or service termination, and even pose severe security threats (Bannister, 2021; Cimpanu, 2019; Jung et al., 2019; Zhong et al., 2020). Thus, timely detection of DBMS bugs is an emerging need.

DBMS bugs can be roughly categorized into crash-related bugs and logic bugs. The crash-related bug occurs when the DBMS abnormally terminates due to a memory error or an assertion failure. A logic bug occurs when DBMS produces an incorrect output for a given test case. Recent works show that coverage-guided mutation-based DBMS fuzzing systems (Andreas Seltenreich, 2022; Chen et al., 2021; Zhong et al., 2020; Liu et al., 2021; Zhang et al., 2021; Ghit et al., 2020) have proven more effective than generation-based ones because they can generate more diverse test cases to trigger the bugs.

However, effectively detecting DBMS bugs has two innate challenges. *The first challenge is how to generate syntactically and semantically correct SQL statements as test cases.* DBMS performs syntactic and semantic checking on the SQL statements to ensure their validity. Invalid SQL statements will be discarded before being fed into the DBMS execution engine and therefore cannot trigger bugs in the execution engine. *The second challenge is how to propose effective oracles to capture logic bugs.* Unlike crash-related bugs that can easily be captured by sanitizers such as ASan (Anon, 2022a), logic bugs that generate incorrect results are hard to detect.

Limitations of Existing Systems Existing DBMS fuzzers have limitations in addressing these challenges. First, current DBMS fuzzers are limited in generating diversity and valid SQL statements due to the incomplete and inaccurate semantic constraints they obtain. In general, a test case contains multiple SQL statements, where each statement contains many variables (i.e., identifiers and constants) that should satisfy the semantic constraints to guarantee semantic correctness. Existing systems always try to strike a balance between the diversity and validity of generated queries. For example, SQLancer (Rigger and Su, 2020a,b,c; Ba and Rigger, 2024) achieves high validity of generating queries but

* Corresponding author.
E-mail addresses: lijiaqi93@zju.edu.cn (J. Li), krking@zju.edu.cn (K. Wang), yaoguang.cyg@antgroup.com (Y. Chen), yajin_zhou@zju.edu.cn (Y. Zhou), lei_wu@zju.edu.cn (L. Wu), 12221251@zju.edu.cn (J. Wang).

https://doi.org/10.1016/j.cose.2025.104564
Received 16 February 2025; Received in revised form 25 May 2025; Accepted 6 June 2025
Available online 7 July 2025

covers only a limited subset of the entire SQL grammars. In contrast, Squirrel (Zhong et al., 2020) and SQLRight (Liang et al., 2022) support full SQL grammars but can only gather identifier types (e.g., table, column) constraints. However, these constraints are incomplete, causing many invalid queries to be generated. For example, even though all identifier types in the statement ''`SELECT Name+1 FROM StudentList`'' are correct, it is invalid in PostgreSQL because it adds a string to an integer. Therefore, we need an SQL statement generator that supports all SQL grammars while maximizing the validity of generated statements to enhance testing efficiency.

Second, oracles used by existing systems (Rigger and Su, 2020c,a,b; Liang et al., 2022; Tang et al., 2023; Hao et al., 2023; Song et al., 2023) to detect logic bugs have strict requirements on the SQL statements. This makes the fuzzing system only explore a narrow space of inputs, leading to limited bugs that can be detected. For example, NoREC (Rigger and Su, 2020a) requires that the effective SQL query in a test case has `WHERE` clauses, thus it can only detect logic bugs due to the optimization in `WHERE` clauses. To address the limitation, TQS (Tang et al., 2023) and MOZI (Liang et al., 2024) leverage the idea of differential testing. They try to explore multiple query plans for SELECT statements by changing the configuration of DBMS and discover logic bugs by comparing their execution results. Each query plan can be viewed as an optimization combination of the DBMS for that statement. However, both of them can only cover partial optimization combinations in DBMS. Hence, an oracle that can be applied to SQL statements without strict requirements and explore more query plans is needed.

Our Solution This work proposes two key techniques to solve the two innate challenges. The first key technique is called *context-sensitive instantiation*, which performs a context-sensitive analysis to collect more comprehensive and accurate semantic constraints to improve the semantic correctness of the generated input. For instance, a SQL statement ''`SELECT x1+i1 FROM x2 WHERE x3=x4`'' has four identifiers `x1~x4` and a constant `i1`. Squirrel and SQLRight can only infer that `x1`, `x3` and `x4` are columns of the table `x2`. In contrast, our system can additionally infer that the type of column `x1` must be numeric and the type of columns `x3` and `x4` should be comparable (e.g., both are string). With more inferred semantic constraints, our system can enhance the validity of generated test cases while maintaining their diversity. We evaluated the effectiveness of our method and found it achieves 1.14x-3.22x higher semantic correctness than previous approaches (Section 5.2).

The second key technique is called *multi-plan execution* (abbreviated as MPE), a novel and general test oracle for detecting logic bugs in DBMS. MPE does not put strict limitations on the SQL queries that can be tested, which improves the capability of edge and bug discovery (Section 5.4). MPE leverages the idea of differential testing to *compare the execution results of multiple query plans of a SQL query*. Specifically, when a query is submitted to a DBMS, the query optimizer typically evaluates various query plans and determines the optimal one to execute. MPE hooks the query optimization process to make the DBMS execute all the query plans (instead of the optimal one) and compares the results. A logic bug is detected if the results from different query plans are inconsistent.

Prototype and Evaluation We implemented a prototype named Kangaroo and applied it to three widely-used DBMSs: SQLite (Anon, 2022j), PostgreSQL (Anon, 2022h), and MySQL (Anon, 2022g), to evaluate its effectiveness. Kangaroo successfully identified 54 new bugs, consisting of 13 logic bugs, 19 crashes, and 22 assertion failures. As of the time of writing, 34 of these bugs have been fixed, with 11 assigned CVEs (Anon, 2022d). These results demonstrate the efficacy of our system. We also compared Kangaroo with leading DBMS testing tools: Squirrel, SQLancer, and SQLRight. After conducting a 24-hour test on the three DBMSs, Kangaroo significantly outperforms these tools embedded with varying test oracles in terms of edge and bug discovery.

This work makes the following main contributions.

```
Expression:   // The following shows five of the patterns Expression has
| Literal        // Constant
| ColumnReference
| Expression EQUAL Expression
| Expression AND Expression
| Expression ILIKE Expression
.....
TableReference:
| JoinedTable
| SingleTableRef
.....
SingleTableRef:
| IDENT            // Identifier
.....
JoinedTable:
| TableReference JOIN TableReference JoinConstraints
.....
JoinConstraints:
| ON Expression
.....
```

Fig. 1. The example illustrates several syntactic rules of SQL. It contains five SQL clauses and some of their patterns. The uppercase words represent SQL keywords or variables, while the others represent SQL clauses.

- We revealed two challenges of effectively detecting DBMS bugs and the limitations of existing solutions.
- We proposed two key techniques to solve the challenges, including context-sensitive instantiation to improve the semantic correctness of the mutated SQL queries and the MPE that can be applied to richer kinds of SQL queries can cover more query plans.
- We implemented a prototype system and applied it to three popular DBMSs and successfully detected 54 new bugs. A comparison with the previous approaches demonstrates the effectiveness of our two key techniques.

2. Background

2.1. Structured Query Language

Relational DBMSs use Structured Query Language (SQL) (Chamberlin and Boyce, 1974) for querying and maintaining the database. SQL statement is the smallest execution unit that typically consists of one or several SQL clauses, e.g., `WHERE` clause, `FROM` clause. Each SQL clause is a meaningful logical chunk that consists of tokens (including identifiers, constants, and SQL keywords), and sub-clauses. *We refer to identifiers and constants as variables in this paper* because their values are changeable and their changes do not affect the syntactic structure.

As shown in Fig. 1, each SQL clause consists of one or more specific patterns, with each pattern corresponding to a particular semantics. For example, the `expression` clause can be a literal, a column, or an arithmetic or logical expression. Each pattern may impose semantic requirements on sub-clauses or variables. For instance, the arithmetic expression pattern typically requires operands to be numeric. When the operands are columns, they must also be numeric. In conclusion, the semantic requirements of a variable (i.e., semantic constraints) are determined by the clauses to which it directly or indirectly belongs, as well as the patterns of these clauses. Failure to meet these requirements will result in a semantic error, causing the DBMS to reject the statement during earlier validation checks.

In this paper, we refer to the semantics of clauses as context information. Since our method derives semantic constraints by analyzing the patterns of all SQL clauses in a statement, we call it *context-sensitive instantiation*.

2.2. SQL query processing

A typical DBMS processes an SQL query in several stages, involving four main components: the parser, translator, planner, and executor. A

Table 1
Eight levels of SQL statement correctness and the corresponding reachable DBMS modules. The final column indicates the theoretical lower bound of the correctness level for input generated by different fuzzers[a].

Level	Input class	Module	Incorrect input examples	Fuzzers
1	Binary input	Lexer	–	AFL
2	Sequence of ASCII	Lexer	Binary input	
3	Sequence of words	Parser	Incorrect SQL keywords	
4	Syntactically correct	Translator	Missing semicolon	
5	Identifier type correct	Translator	Query a trigger	Squirrel SQLRight
6	Data type correct	Translator	Add an integer to a string	
7	Statically conforming	Executor	Ambiguous column names	**Kangaroo**
8	Dynamically conforming (Semantic correctness)	Executor	Scalar query returns multiple values	

[a] Fuzzers that do not support the full SQL grammar, such as SQLancer, are excluded from the table.

typical DBMS consists of four main components: the parser, translator, planner, and executor. First, the parser performs lexical analysis to divide the query into tokens (e.g., SQL keywords). It then performs syntax analysis to construct a raw parse tree that represents the query's syntactic structure. During this process, it checks for syntax validity and terminates execution if any errors are found. Next, the translator analyzes the parser tree, checks its semantics, and converts it into an internal representation that can be used by the optimizer and the executor. The planner (also called optimizer) will try to find all possible query plans for a SQL statement and select the optimal one by evaluating the cost of each query plan. Finally, the executor of DBMS runs the optimal query plan and returns the required result. The executor also checks whether the dynamic semantics are correct during execution.

Among these components, the planner and executor are the most complex and error-prone. Therefore, improving the semantic validity of generated queries is essential to effectively testing these two components.

2.3. Levels of DBMS testing

The correctness of a SQL statement can be divided into eight levels as shown in Table 1. For a syntactically correct SQL statement, the value of variables in the statement determines whether the statement is semantically correct. The statement's context information imposes various restrictions, including *static constraints* checked by the translator and *dynamic constraints* checked by the executor. For example, in the statement `SELECT 1==(SELECT a FROM t1), a` should be numeric, which is a static constraint; the subquery `SELECT a FROM t1` should return no more than one row, which is a dynamic constraint. Statements with all constraints satisfaction are semantically correct (level-8). A statement is statically conforming (level-7) if it meets all static constraints but fails to satisfy certain dynamic constraints. Static constraints can be further divided into different levels. Identifier type correct (level-5) indicates that all identifiers in SQL statements have correct types (e.g., Column) and the data type correct (level-6) guarantees the data type (e.g., integer) of variables is correct.

Different systems can guarantee different correctness levels. General-purpose fuzzers (e.g., AFL (Anon, 2022c)) are inefficient for DBMS testing since they lack sophisticated approaches to generate higher-level correct inputs. Benefiting from syntax-preserving mutations and constraint-guided instantiation, Squirrel is more likely to generate higher quality input, thus significantly improving the ability to detect bugs. However, Squirrel has to limit the complexity of generated queries to tolerate its incomplete and inaccurate constraints. To better explore the core modules of DBMS, we propose a new method to generate more diverse and complex test cases, while at the same time, they can be statically conforming (level-7).

2.4. Test case generation

DBMS testing aims to trigger bugs by constructing abundant test cases that typically contain multiple SQL queries. The query generation consists of two phases: **structure generation** and **variable instantiation**.

The structure generation constructs the syntactic structure of SQL queries (i.e., without concrete variables) to ensure the generated SQL queries can pass the SQL parser. There are two structure generation methods. The rule-based ones (Sutton et al., 2007; Tang et al., 2023; Rigger and Su, 2020a,b,c) generate test cases following a predefined model. However, building a precise model requires domain knowledge. Besides, the generated inputs cannot efficiently explore the program's state space since they waste much effort on similar queries. Mutation-based methods (Neystadt, 2008; Zhong et al., 2020; Liang et al., 2022, 2023; Fu et al., 2022) generate new test cases by performing grammar-based mutation on seed queries. Specifically, it first generates a syntax tree for seed queries, then creates specific mutations by using mutation operators on a tree node.

The variable instantiation concretizes the variables within queries to improve the semantic correctness of generated SQL queries. For a given syntactic structure of SQL queries, each variable within SQL queries should satisfy specific constraints to ensure semantic correctness. The variable instantiation analyzes the syntactic structure of SQL queries to collect the constraints, and then fills the variables with concrete values that satisfy these constraints. However, inferring comprehensive and accurate constraints from SQL statements is challenging due to the complexity and diversity of SQL grammar. Without accurate constraints, previous DBMS fuzzers tend to generate many invalid queries. To tolerate the inaccurate constraints, previous works either support a small subset of SQL grammars (Rigger and Su, 2020a,b,c) or limit the complexity of generated queries (Zhong et al., 2020; Liang et al., 2022).

2.5. DBMS test oracles

An oracle is a mechanism for determining whether the actual outputs match the expected outcomes, facilitating logic bug detection. The two most widely used types of test oracles are differential testing and metamorphic testing. Differential-testing-based oracles (Slutz, 1998; Jung et al., 2019; Liang et al., 2024) use different implementations of the same functionality as cross-referencing oracles. They provide the same inputs to a series of similar systems, such as DBMSs with different configurations, and then observe the results. Any inconsistency between the results may indicate a potential bug. However, these methods typically test limited DBMS functionalities. For instance, functionalities that cannot be modified through configuration settings remain untestable when comparing DBMSs with different configurations.

Metamorphic-testing-based oracles (Rigger and Su, 2020a,b; Ba and Rigger, 2024; Hao et al., 2023) address the test oracle problem by applying a specific input transformation that produces predictable changes in the output. They detect bugs by verifying whether the transformed query's output exhibits the expected change compared to the original query's output. For example, NoREC (Rigger and Su, 2020a) converts a query that is potentially optimized by DBMS into a semantically equivalent one that can hardly be optimized, then checks for consistency between the results of the original and the translated query. However, these methods are typically limited to a subset of SQL that can be translated, thereby limiting the scope of queries that can be tested.

3. Motivating examples

In this section, we use two real examples to demonstrate the advantages of our system's two key techniques.

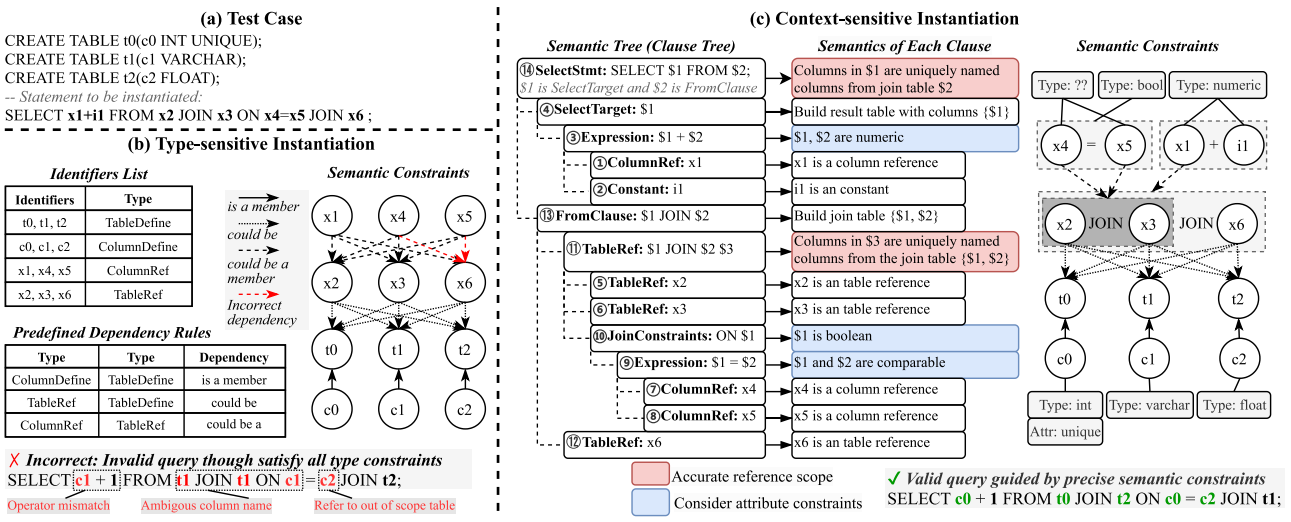

Fig. 2. Comparison of Variable Instantiation Approaches. (a) A mutated test case with the final statement undergoing instantiation. (b) Type-sensitive instantiation relies on predefined rules and identifier types, often producing incorrect and incomplete constraints. (c) Context-sensitive instantiation constructs a semantic tree and performs bottom-up analysis to infer accurate and complete constraints, improving the validity of generated queries. Each node in the semantic tree represents a SQL clause and its associated pattern, with child clauses indexed as $1, $2, etc. The leading number denotes the order of bottom-up analysis.

3.1. Context-sensitive instantiation

As aforementioned, variable instantiation requires that the variables in the SQL statement meet their semantic requirements to ensure semantic correctness. However, due to the significant syntactic differences between DBMSs, using a general approach to infer accurate semantic constraints from SQL statements is challenging. Due to the limited human effort, some test case generators in previous DBMS fuzzers (Rigger and Su, 2020a,b,c; Ba and Rigger, 2024), only support a small subset of SQL grammars, which limits the diversity of the queries they generate. Therefore, the input space that can be tested is narrow. Expanding this approach to support full SQL grammars requires significant engineering effort, and the grammar differences across DBMSs limit its universality.

To address this issue, some DBMS fuzzers (Zhong et al., 2020; Liang et al., 2022) use a general method (called type-sensitive instantiation in this paper) that infers the dependency between identifiers (one type of semantic constraint) through theirs type (e.g., `TableDef`, `ColumnRef`) and some manually predefined rules. However, without the context information, these fuzzers tend to build incorrect and incomplete dependencies among variables, causing many invalid queries to be generated. Moreover, the difficulty of guaranteeing query validity dramatically increases as query complexity grows. To mitigate this impact, they limit the complexity of generated queries by limiting the length of statements in test cases and the size of test cases. Even so, they still generate over 40% invalid queries.

Generating invalid queries has a significant negative impact on fuzzing performance. First, invalid queries discarded by DBMS earlier validation checks cannot trigger deep logic, such as optimization and execution processes. Therefore, obtaining accurate semantic constraints to improve the semantic correctness of generated queries is critical for detecting deep bugs in the DBMS. More importantly, invalid queries are ineffective for logic bug detection, as they cannot be executed successfully and produce output for comparison.

Why Existing Works Suffer from Inaccurate Semantic Constraints. Squirrel (Zhong et al., 2020) proposes type-sensitive instantiation which can instantiate queries involving full SQL grammars supported by DBMS at the cost of lower validity. Specifically, it labels the types of identifiers during parsing, and infers their dependencies according to manually predefined rules. Then, it instantiates variables to ensure that they satisfy all these dependency constraints. For example, the third dependency rule in Fig. 2b builds the dependency constraints that any identifier of type `ColumnRef` could be a column of any identifier of type `TableRef`. Based on type-sensitive instantiation, SQLRight (Liang et al., 2022) enhances the accuracy of inter-statement constraints by maintaining DBMS states (e.g., table schema). This prevents statements from referencing tables that have been dropped by a preceding DROP statement. These approaches do not require consideration of context information, making their implementation simple and ensuring their universality across various DBMS syntax implementations.

However, all these works suffer from **incorrect and incomplete intra-statement constraints**, as they only infer the dependency according to predefined rules but ignore the context information. The inaccurate constraints cause many invalid queries to be generated. Fig. 2b shows an incorrect concrete statement suffers from three different semantic errors even though it satisfies all constraints considered by type-sensitive instantiation utilized by SQLRight.

The **incorrect constraints** stem from assigning the scope of each identifier based on predefined, coarse-grained rules, which are inherently inaccurate, such as assigning the scope of an identifier to the statement it belongs to. In Fig. 2b, they build an incorrect constraint that `x5 could be a member of x6` based on the dependency rules that a `ColumnRef` could be a member of any `TableRef`. Unfortunately, because `x4` belongs to the `TableReference x2 JOIN x3 ON x4=x5`, `x4` could only be the member of `x2` or `x3`.

The **incomplete constraints** mainly result from ignoring the attribute requirements of the identifiers. For example, a column identifier may need to meet certain requirements in terms of name, data type (e.g., INT, TEXT), and attributes (e.g., `PRIMARY KEY`, `FOREIGN KEY`, `GENERATED`), etc. In Fig. 2b, the invalid query suffers from ambiguous column names and illegal expression errors due to the failure to account for the constraints on the name and data type of columns. Furthermore, the **incomplete constraints** may also arise from overlooking the semantic requirements on constants. For example, the semantics of a constant might represent the ordinal position of a column in a table, so its valid values should be integers between 0 and the number of columns in the table.

Extending the predefined dependency rules further can help mitigate the false positives and false negatives in constraints derived from type-sensitive instantiation. However, the absence of context information hinders the accurate assignment of scopes to identifiers and the recognition of their attributes and associated requirements. As a result,

it cannot fundamentally resolve the problems of false positives and false negatives.

Our Solution We propose context-sensitive instantiation, which infers constraints of variables, including both identifiers and variables, from their contextual information (i.e., the SQL clause pattern and relationships among clauses), rather than predefined rules. Due to differences in grammar implementations across DBMSs, deriving semantic constraints from context presents significant challenges to generalizability. To overcome this, as shown in Fig. 2c, we first convert the syntax tree into a unified semantic tree, where each node corresponds to a specific SQL clause. Semantic constraints are then inferred by traversing this semantic tree and analyzing each clause node in a post-order manner.

This approach enables the accurate capture of attribute requirements and scopes for variables, thereby avoiding the incorrect and incomplete constraints that type-sensitive strategies encounter. Specifically, in the `Expression` clause ③, we can infer that both the column `x1` and the constant `i1` must be numeric, preventing the operator mismatch error. In the `TableReference` clause ①, the context restricts `x4` and `x5` to reference only columns within the joined table `x2 JOIN x3`, thereby preventing erroneous references to `x6`. Besides, this `TableReference` clause also enforces that `x4` and `x5` must reference unique columns, which prevents self-joins in this clause.

Since this process is similar to the semantic checking process in DBMSs (DBMSs check the semantics during tree traversal whereas we collect the constraints they check), it can theoretically obtain accurate static constraints and thereby generate statically conforming statements. Our approach ignores dynamic constraints because enforcing dynamic constraints requires intermediate states during query execution, which is challenging. We leave it as part of future work.

3.2. Multi-plan execution

Metamorphic testing is a prevalent approach to testing DBMS. It finds some specific changes or transformations to the input that will cause predictable changes to the output. A result that does not conform to the expected changes indicates a potential logical bug. However, all these oracles (Rigger and Su, 2020a,b; Hao et al., 2023; Song et al., 2023; Ba and Rigger, 2024) construct test cases based on specific features, which limits the SQL grammar they can support. For example, NoREC (Rigger and Su, 2020a) requires the queries to be a SELECT statement with `WHERE` clause such that it can do the transformation. Another oracle PQS (Rigger and Su, 2020c) generates queries along with its ground-truth result which also puts strict limitations on generated queries.

Another type of DBMS test oracle leverages the idea of differential testing. TQS (Tang et al., 2023) and MOZI (Liang et al., 2024) run one query under different DBMS optimization settings, which may alter the query plans chosen for execution. This method imposes fewer restrictions on test queries, allowing it to test a wider range of DBMS functionality. However, optimization settings in DBMS typically only manipulate a subset of the optimizations, therefore some optimization strategies in DBMS can never be tested by this approach. In addition, optimization settings cannot force optimization adoptions if the optimization cannot reduce the amount of computation. Besides, optimization settings are global settings that cannot control the scope of optimizations. For example, it cannot disable the index search in table `t1` but enables index search in `t2`. These limit their ability to explore various combinations of optimizations, thus covering limited query plans.

We propose a novel test oracle, MPE, which imposes no constraints on SQL grammar and can explore all possible query plans for each query. Specifically, it hooks into the DBMS optimizer to execute all query plans and compare their results. If the results of these query plans are inconsistent, a logic bug is detected. In the following, we use one real-world SQLite bug in Fig. 3 to illustrate why the existing oracles

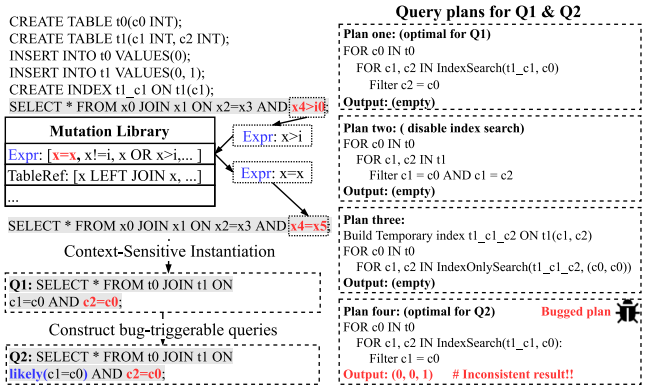

Fig. 3. A mutated test case that exposes a real-world logic bug in SQLite, detected by MPE but missed by previous oracles (e.g., NoREC, TQS). The instantiated SELECT statement Q1 yields four valid query plans, with the last one producing a result inconsistent with the others. By manually transforming Q1 into Q2, we construct a query that can trigger the buggy plan in the original DBMS.

cannot detect the logic bug and how the MPE can capture such a bug. When the test case is executed, the DBMS optimizer generates four valid query plans for the `SELECT` statement Q1. Among these, plan one is the optimal plan and is executed by SQLite by default, while plan four is the buggy plan.

Why Existing Works Cannot Detect the Bug. The oracles NoREC, TQS, and MOZI all attempt to explore different query plans for comparison. Specifically, NoREC shifts the condition in the `WHERE` clause to the `SELECT_TARGET`, then executes both to compare the execution result. However, the SELECT statement Q1 in Fig. 3 does not meet the requirement (lacks `WHERE` clause), thus unable to detect the bug. TQS and MOZI can additionally test plan two in Fig. 3 by turning off index optimization. However, they still fail to reveal the bug, which only occurs in plan four.

Why Our System Can Capture the Bug. By hooking the query plan selection function, our system executes all valid query plans for a given query and compares their outputs. Since plan four produces a non-empty result that differs from the results of the other three plans, our system successfully identifies this as a logic bug.

One may argue that the bug found in query plan four is meaningless since it will not be executed by default. In other words, the bug cannot be triggered in practice. However, even though the buggy query plan is not optimal for the `SELECT` statement in the test case, it can be the optimal one by adjusting the statement or changing the optimization setting. For example, by adjusting the statement Q1 to Q2, we successfully triggered the buggy query in the original DBMS. Actually, almost all bugs detected by MPE can be triggered in practice. We discuss exceptions later in Section 6.2.

The root cause of this bug is the incorrect equivalence transfer optimization. Because the expression in `join constraints` clause is `c1=c0 AND c2=c0`, which implies `c1=c2`. SQLite uses the index on `c1` for the constraint on `c2` because of this inference. Specifically, it uses the values of `c0` as the key to search index `t1_c1` to fetch all records that satisfy `c1=c0` and then tries to judge `c1=c0` on these records again. However, it should check `c1=c2` rather than `c1=c0`. As a result, the expression `c1=c0 AND c2=c0` is incorrectly optimized to `c1=c0`.

3.3. Focus of this paper

This work aims to enhance the effectiveness of fuzzers in detecting bugs, especially logical ones, in DBMSs by proposing two novel techniques. The first technique is a variable instantiation (Section 2.4)

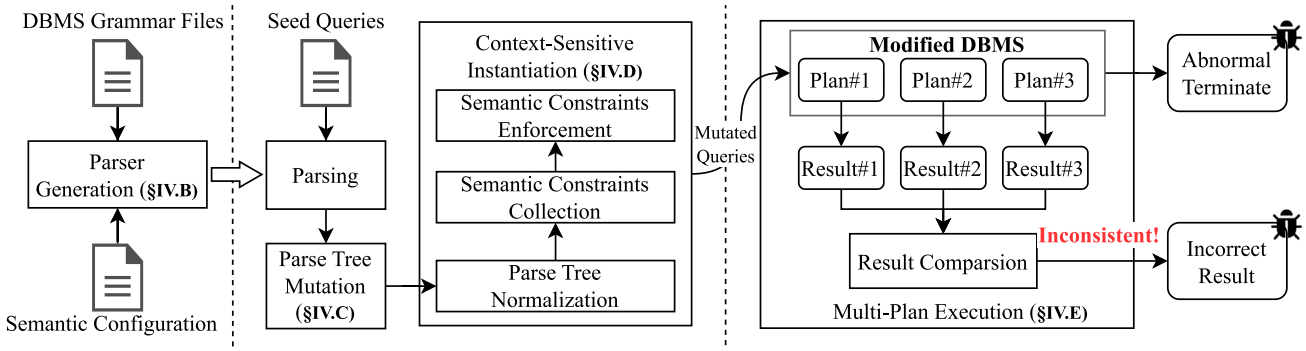

Fig. 4. The overall architecture of our system.

approach, which infers more accurate semantic constraints on statements to improve the semantic correctness of generated SQL statements. Existing works that propose new SQL structure generation approaches (Neystadt, 2008; Zhong et al., 2020; Liang et al., 2022, 2023; Fu et al., 2022; Wang et al., 2019) or utilize error feedback to guide the variable instantiation (Jiang et al., 2023) are orthogonal to our proposed technique. The second technique is a test oracle that can detect logic bugs on a broader range of DBMS functionality than previous approaches.

4. Design

4.1. Overall design

Our system aims at detecting both crash-related bugs and logic bugs in DBMSs. Fig. 4 illustrates the overall architecture of our system. The *parser generation* (Section 4.2) takes a semi-automatic way to generate SQL parsers before the testing. During the testing, it first performs a *parse tree mutation* (Section 4.3) to mutate the query structure while preserving syntactic correctness. Then, our system conducts the process of *context-sensitive instantiation* (Section 4.4) to instantiate all variables in queries. At last, the mutated SQL queries will be executed by the DBMS. The *multiple-plan execution* (Section 4.5) hooks into the optimizer in DBMS to execute all query plans (instead of the optimal one) and compare the returned results. Any inconsistency between the results indicates a logic bug. Any crash-related bug can also be detected during the execution of query plans.

Example We use the test case in Fig. 3 that triggers a logic bug in SQLite to illustrate the workflow of our system. Generating this test case takes several rounds of mutation on the initial seed. We take the last round of mutation as an example. The last round of mutation is performed on the `SELECT` statement which replaces the expression node `x>i` with the expression node `x=x`. After that, our system performs context-sensitive instantiation to replace the symbolic values with concrete values. Specifically, it first normalizes the parse tree into a unified semantic tree to eliminate the differences in syntactic structure. Then, it analyzes the semantic tree to collect semantic constraints on variables. Next, it utilizes the randomized backtracking algorithm (Anon, 2023a) (also called randomized depth-first search) to find a random solution for all variables that satisfy all semantic constraints. At last, it translates the concretized semantic tree to an SQL string as a new statement (Q1). The mutated test case will be fed to the modified DBMS which executes all four query plans one by one and finds that three of them return an empty result, while the fourth plan returns a non-empty value. This leads to the detection of the logic bug.

4.2. Parser generation

The syntactic rules of SQL parsers vary in different DBMSs. A valid statement for SQLite may be rejected by the parser of PostgreSQL. To guarantee the mutated test case is syntactically correct, an accurate parser for each DBMS is needed.

Implementing an accurate parser requires non-negligible effort. To save manual effort, we propose a semi-automatic approach to generate the parser. Our core insight is that almost all DBMSs leverage a standardized format called Backus Normal Form (BNF) (Knuth, 1964) to describe the grammar of the supported SQL queries. Thus, we can construct an accurate parser by automatically extracting and analyzing the DBMS BNF rules. Specifically, the parser generator extracts syntactic rules (Fig. 1) from the target DBMS grammar file. Then, it uses syntactic rules and a semantic configuration to build the parser. We explain the semantic configuration in parse tree normalization of Section 4.4.

4.3. Parse tree mutation

Our system mutates the SQL queries based on the parse tree. This can ensure the mutated queries are always syntactically correct. Similar to Squirrel (Zhong et al., 2020), we also focus on structure mutation since it is more effective than data mutation. Specifically, our system symbolizes all the variables in statements and concretizes them after the mutation.

To mutate a test case, our system randomly picks up one node v from the parse tree and randomly fetches a new one w with the same type from the mutation library. Then, it replaces v (including its children) with w. The mutation library is a dictionary where the key is the node type and the value is a list of distinct parse trees rooted at nodes of that type. Our system accepts DBMSs' official test cases to initialize the mutation library. The mutation strategy is not a contribution of our work, as it is basically the same as the previous work (Zhong et al., 2020).

4.4. Context-sensitive instantiation

After mutating the parse tree, we analyze the SQL statement to collect semantic constraints and instantiate all variables in the mutated query. Building a general analysis module for SQL is challenging for several reasons. First, the query analysis logic in DBMSs is usually deeply embedded and difficult to reuse or extend. Second, although DBMSs typically follow the standard SQL grammar, they often include many extra clauses. Some of these are *dialect clauses* (e.g., hint clauses) used to support DBMS-specific features. Others are *internal clauses* introduced during implementation. These internal clauses do not contribute to actual meaning but increase the complexity of constraint analysis. More importantly, these internal clauses vary across DBMSs, making it difficult to design a unified approach for constraint analysis.

To address these challenges, we normalize the parse tree into a unified semantic tree. Each node in the semantic tree represents a SQL clause. This unified representation allows the semantic tree to capture the meaning of SQL statements in a simpler and more general way. To support different DBMS dialects, the semantic tree grammar

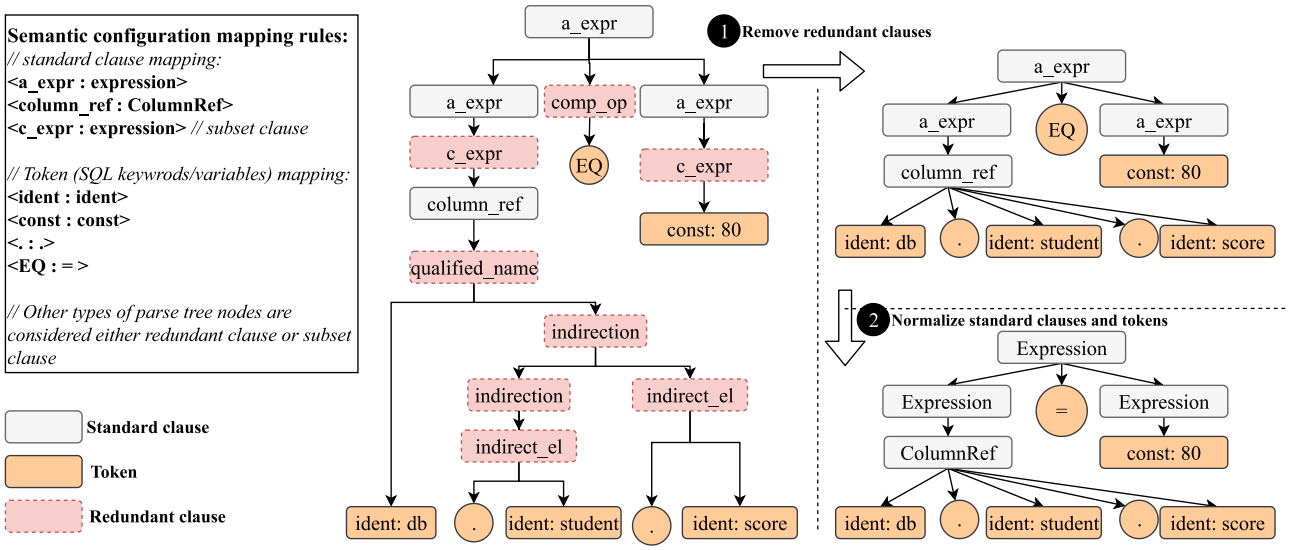

Fig. 5. An example to show the process of translating a parse tree to a unified semantic tree.

includes both standard clauses (e.g., `WHERE` clause) and dialect clauses. We then perform context-sensitive analysis on this semantic tree to extract semantic information and infer constraints. Finally, we use these constraints to guide variable instantiation.

Step I: Parse Tree Normalization The parse tree is a hierarchical representation of the syntactic structure of an SQL statement, where each node corresponds to a clause or token in the statement. However, different DBMSs often produce significantly different parse trees for the same SQL input due to variations in their parsers. To enable generalizable semantic analysis, we normalize parse tree into a unified semantic tree that abstracts the SQL clause structure in a consistent manner.

Parse tree differences across DBMSs stem from two main factors. **First**, DBMSs introduce extra clause definitions in their parsers. These include *dialect clauses* and *internal clauses*. *Dialect clauses* support DBMS-specific features. For example, MySQL introduces a `hint` clause to guide query optimization. We retain such clauses in the semantic tree to preserve compatibility with DBMS-specific features. *Internal clauses* are used for implementation convenience but do not contribute to actual meaning. As such, they are considered *redundant clauses* and are removed during normalization. For instance, some DBMSs define a `comp_op` clause to unify comparison expressions as `expression comp_op expression`. Removing this clause yields a more granular form, such as `expression '>' expression`, without semantic loss. More importantly, these granular patterns tend to be more consistent across different DBMSs. **Second**, clause and token names (including SQL keywords and variables) are often inconsistent across DBMSs. To resolve this, we manually annotate each clause with its corresponding standard SQL clause name in semantic configuration.

Based on these insights, we design a two-step normalization procedure (Fig. 5) to produce a normalized semantic tree for downstream tasks. (1) Remove *redundant* clauses from the parse tree. For each removed clause, its child nodes are directly reattached to the parent. (2) Convert the tokens and remaining clauses and tokens into corresponding semantic clause according to the semantic configuration.

Step II: Semantic Constraints Collection We analyze the semantic tree to infer semantic constraints. Algorithm 1 illustrates the constraint inference process. For each SQL statement, the system conducts a post-order traversal of its semantic tree. During traversal, each semantic node (i.e., SQL clause) is analyzed to identify its pattern. Based on the identified pattern, the corresponding `GetConstraint` function is invoked to extract semantic information and infer relevant constraints. These `GetConstraint` functions are implemented according to the target DBMS's official documentation. This design enables precise inference of variable scopes and attribute requirements.

SQL clauses with the same syntax may pose different semantics constraints across DBMSs. To address this, we provide override functions to accommodate the specific requirements of each DBMS. This design ensures efficient adaptation of our system to new DBMSs with minimal additional effort. A detailed description of the `GetConstraint` implementation is provided in Appendix A.1.

Due to the diversity and volume of SQL clauses, implementing customized constraint analysis for each clause requires substantial engineering effort. To balance accuracy with generality and reduce implementation overhead, we adopt a hybrid granularity semantic analysis strategy. Specifically, for clauses not yet covered by fine-grained customized analysis, a coarse-grained but generic semantic analysis is applied as a fallback. This generic analysis is similar to type-sensitive instantiation, which infers approximate semantic constraints based on predefined rules. This design allows for direct adaptation to new DBMSs and progressive refinement as fine-grained analysis is incrementally developed.

Step III: Semantic Constraints Enforcement Conceptually, enforcing semantic constraints can be viewed as a Constrained Random Sampling (CRS) problem. It involves finding a solution to a constrained problem such that each feasible solution has an approximately equal probability of being chosen. Although the problem of finding a single solution to constrained problems has been extensively studied, less research has been paid to the efficient generation of random solutions. Given the relatively low complexity of static semantic constraints in SQL queries, our system adopts a randomized backtracking algorithm (Anon, 2023a) to solve these constraints. This method allows us to generate valid solutions while preserving randomness. In future work, we intend to integrate more sophisticated constraint sampling techniques, such as SMTSampler (Dutra et al., 2018), to efficiently find more diverse solutions.

The high-level idea of randomized backtracking algorithm is to randomly enumerate all possible dependencies for each variable until finding one determined dependency that satisfies all constraints. Initially, all variables are unassigned. At each step, a variable is chosen and a random candidate value is assigned to it and the satisfaction of the partial assignment is checked. If the partial assignment is valid, the algorithm proceeds recursively with the remaining unassigned variables. If a conflict is detected, it performs backtracking to explore alternative assignments. A valid solution is found when all variables

Algorithm 1: Algorithm of semantic analysis.

Input: SemanticTrees: the root node of the semantic tree of an SQL statement
Output: ConstraintSet: all semantic constraints of the current statement

```
1  Function CollectConstraints( SemanticTree):
2      ConstraintSet ← vector()
3      PostOrderTrav( SemanticTree, ConstraintSet)
4      return ConstraintSet
5  Function PostOrderTrav( SemanticNode,
     ConstraintSet):
       // children of the semantic node include
          semantic node, SQL keyword, and variables
6      for each child C in SemanticNode.children do
7          if C is semantic node then
8              PostOrderTrav(C)
9          end
10     end
11     constraints ← ClauseAnalyze (C)
12     ConstraintSet.update( constraints)
13 Function ClauseAnalyze( SemanticNode):
14     MatchPattern ← FindPattern(
       SemanticNode.children) // find the matching
       pattern of the SQL clause
15     constraints ← Getconstraint( MatchPattern)
16     return constraints
```

```
1  @@ -5496,6 +5607,9 @@ get_cheapest_fractional_path(
       RelOptInfo *rel, double tuple_fraction)
2          Path       *best_path = rel->
               cheapest_total_path;
3          ListCell   *l;
4
5  +       if (multi_plan_enable)
6  +               return select_next_plan(rel, plan_id);
7
8          /* If all tuples will be retrieved, just return
               the cheapest-total path */
9          if (tuple_fraction <= 0.0)
10                 return best_path;
```

Listing 1 A patch snippet for PostgreSQL illustrating how to control the plan choose function to execute all query plans. The `get_cheapest_fractional_path` is the plan choose function, and we hook into it by adding two lines of code (lines 5–6). When multiple execution plans are enabled, it directly returns the execution plan with the ordinal number `plan_ordinal`. We re-execute the queries with different `plan_ordinal` until all query plans have been executed.

have been successfully assigned values that collectively satisfy the full set of constraints.

Contribution Summary Context-sensitive instantiation introduces a novel analysis approach to extract static semantic constraints from SQL statements. Unlike prior approaches that rely on manually predefined rules, it infers variable constraints based on the contextual structure of SQL clauses. This effectively addresses the issues of inaccurate and incomplete constraints in type-sensitive instantiation. By capturing more precise and comprehensive constraints, it significantly improves the validity of generated SQL test cases.

Context-sensitive instantiation is generalizable across different DBMSs. First, it resolves structural differences in parse trees generated by different DBMSs for the same SQL statement by normalizing them into a unified semantic tree. All dialect-specific clauses are preserved within the semantic tree to support further semantic analysis. Besides, it adopts a hybrid granularity strategy to guarantee both scalability and compatibility with diverse SQL dialects. More specifically, it combines coarse-grained analysis for compatibility with fine-grained analysis for precise constraint inference. Developers can gradually implement fine-grained analysis for dialect clauses to progressively improve the accuracy of constraints.

Context-sensitive instantiation is practically applicable to industrial DBMS. To the best of our knowledge, PostgreSQL and MySQL are among the most complex open-source DBMSs, with approximately 1.05 million and 3.25 million lines of code (LOC), respectively. Our system prototype has been successfully applied to both of them, illustrating the practical applicability of our approach.

4.5. Multi-plan execution

MPE focuses on detecting logic bugs in the planner and executor because these components are the most complex ones and have not been well-tested. Our system hooks the query optimization process to *execute all the query plans for a* SELECT *statement to detect the incorrect one(s) that may cause incorrect results*. Aligned with the previous works (Rigger and Su, 2020a,b,c; Liang et al., 2022; Tang et al., 2023), we choose SELECT statements to demonstrate MPE. MPE can also be used to detect logic bugs in other statements (e.g., INSERT, UPDATE, etc.), which will be explored as part of future work.

Adopting MPE to DBMSs is not straightforward. The main challenge is how to make a DBMS execute all query plans with minor modifications. Our key observation is that DBMS typically employs a plan choose function to compare the estimated cost of different query plans and choose the best one. Therefore, by hooking into the plan choose function (as shown in Listing 1), we can execute any query plan generated by the DBMS. Additionally, we added a loop before the entry function for query processing to continuously execute the statement with different query plans until all plans are executed. Specifically, we record the total number of query plans generated by the DBMS during query plan generation phase. An iterator is then employed to specify the query plan to be executed in each iteration, incrementing sequentially until all plans have been processed. Furthermore, some DBMSs perform pruning during the generation of query plans to discard plans that are deemed inefficient based on cost estimates. This pruning is typically implemented through a dedicated pruning function. By hooking into this pruning function, we can retain execution plans that would otherwise be discarded.

Such modifications have two advantages. First, they do not break the functionality of DBMS, such as generating incorrect query plans, as they only intervene in the query plan selection and pruning process. Second, they require little implementation effort.

Runtime Overhead Executing all valid query plans for a given SQL statement can reduce the number of SQL statements executed within a fixed time budget. However, different SQL statements may compile into the same underlying query plan, resulting in redundant executions. Therefore, fuzzing efficiency is more accurately measured by the number of distinct query plans exercised or by overall path coverage.

From this perspective, the additional plans executed by MPE do not necessarily incur overhead. The actual overhead depends on whether executing multiple plans per statement introduces more redundant execution than only executing the optimal plan. Our ablation study in Section 5.4 demonstrates that enabling MPE slightly improves code coverage and results in the detection of more bugs. These findings suggest that MPE maintains, and potentially enhances, overall fuzzing efficiency despite its increased per-statement cost.

To further improve fuzzing efficiency, we plan to reduce redundant executions by selectively executing query plans in future work. Specifically, each query plan will be uniquely identified via hashing and tracked with an execution count. Previously untested plans will always be executed, whereas previously tested plans will be executed less frequently as their execution count increases. This selective execution

Table 2
The lines of code of different components.

Component	Language	Lines of code
Parser generator	Python	1917
Parse-tree mutator	C++	849
Context-sensitive instantiation	C++	12,476
Result comparison	C++	909
Fuzzer	C	5241
Other	–	798
Total	–	22,190

strategy preserves the benefits of MPE while mitigating unnecessary redundant executions.

Contribution Summary We present MPE, a novel test oracle for detecting logic bugs in SQL statements that contain multiple query plans. Similar to existing oracles such as NoREC and TQS, which detect bugs by comparing different query plans, MPE is not applicable to queries that yield only a single plan.

MPE introduces a new methodology for comparing query plans. Rather than rewriting queries (NoREC) or toggling DBMS optimization settings (TQS), MPE modifies the DBMS by hooking into its plan selection function. This approach enables the comparison of results from all valid query plans for a given statement. MPE offers the following improvements:

Broader SQL feature coverage: Unlike NoREC and TQS, MPE imposes no restrictions on the SQL grammar, allowing it to support a wider range of SQL features. In contrast, NoREC is limited to queries that can be rewritten into a non-optimizable form, and TQS is confined to join queries, primarily targeting join-related optimization bugs.

Improve plan coverage and reduce false negatives: MPE explores a larger set of query plans per statement, enhancing code coverage and increasing the likelihood of exposing bugs hidden in rarely executed plans. Moreover, by comparing results across more valid query plans, MPE mitigates false negatives that occur when the same incorrect result arises in multiple plans due to shared erroneous logic.

4.6. Prototype implementation

We have implemented a prototype system called Kangaroo and applied it to three widely-used DBMSs: SQLite, PostgreSQL, and MySQL. Our system is built on top of AFL 2.56b (Anon, 2022c). It consists of 21.9k lines of code (LoC) in total. Table 2 summarizes each of the components.

The SQL language comprises over 100 distinct types of clauses. Some clauses are standard SQL clauses, and some are dialects. Each clause typically has multiple syntactic patterns, each associated with distinct semantics. For example, the `ColumnConstraint` clause defines the attribute of a column, which contains nine different patterns, e.g., `PRIMARY KEY` and `FOREIGN KEY`.

Due to engineering constraints, our prototype currently supports fine-grained semantic constraint inference for the primary patterns of the 45 most frequently used SQL clauses. Among these, 42 are shared by the three target DBMSs, while the remaining 3 are DBMS-specific. To normalize clause names and facilitate constraint inference, our prototype defines 65 mapping rules for PostgreSQL, 79 for MySQL, and 59 for SQLite in the semantic configuration. Table 3 summarizes the types of semantic constraints extracted from supported clauses.

For unsupported patterns and clauses, as discussed in Section 4.4, our system adopts a coarse-grained but general schema to infer approximate constraints. Although these constraints may be incomplete, the overall semantic constraints captured by our system are always more accurate and comprehensive than those of existing methods.

Table 3
Constraint types supported by our prototype.

Constraint types		Explains
Semantic Type		Semantic types of identifiers and constants
Dependency		Dependencies between identifiers
ColumnAttributes	Name	Column names
	DataType	Data types of columns, such as INT, TEXT
	DataValue	Constraints on column values
	Others	UNIQUE, PRIMARY_KEY, FOREIGN_KEY, GENERATED, NOT_NULL, etc.
TableAttributes	Name	Table names
	StorageEngine	Storage engine of the table
	Collation	Table collation, which affects compression, sorting, and storage of text columns
	Others	TEMPORARY, INSERT, AUTO_INC, MAX_ROWS, etc.
IndexAttributes	Name	Index names
	IndexType	Types of indexes (e.g., B-tree, full-text indexes)
	Others	TEMPORARY
ConstantAttributes	DataValue	Constraints on constant values

5. Evaluation

In this section, we answer the following research questions to show the advantages and effectiveness of Kangaroo.

- **Effectiveness of detecting bugs in real-world DBMSs.** How effective is Kangaroo in discovering new bugs in real-world production-level DBMSs (Section 5.1)?
- **Generating valid queries.** How effective is context-sensitive instantiation in generating valid queries (Section 5.2)?
- **Comparison with existing techniques.** How effective are our two techniques in detecting bugs (Section 5.3)?
- **Benefits of the proposed two key techniques.** How do our two techniques help detect bugs (Section 5.4)?

Experimental Setup We perform all the experiments on a computer with Ubuntu 18.04 system, Intel Core i7-7700, and 32 GB RAM. We enlarge the bitmap size to 512K bytes to mitigate path collisions (Gan et al., 2018). Since SQLancer requires the particular SQLite version 3.34.0, we use this version for comparison. For other DBMSs, we use the latest version, i.e., PostgreSQL version 14.2 and MySQL version 8.0.29.

We compare Kangaroo with three state-of-the-art and open-source DBMS testing tools: Squirrel (Zhong et al., 2020), SQLancer (Rigger and Su, 2020a,b,c), and SQLRight (Liang et al., 2022). Squirrel is a coverage-guided fuzzer that focuses on crash-related bug detection. To eliminate the impact of inconsistent SQL grammars supported, we extend its features to align with Kangaroo's to build Squirrel+. SQLRight is a coverage-guided fuzzer embedded with two test oracles, NoREC and TLP, to detect logic bugs. SQLancer is a generation-based fuzzer for logic bug detection. It supports one more oracle, PQS, compared to SQLRight. We feed the same test cases to all fuzzing systems (except SQLancer) as the initial corpus and provide the same queries to initialize their mutation libraries. SQLancer is a generate-based tool that does not require any initial inputs. We launch five fuzzing instances for each system and run each instance for 24 h. We report the average result except for the bug number. We collect all bug reports from the five fuzzing instances as the final result and then count their first occurrence time for each unique bug.

5.1. Detecting bugs in real-world DBMSs

As shown in Table 4, across intermittent runs during a 20-month period of development, Kangaroo successfully discovered 54 unique

bugs, including 41 crash-related bugs and 13 logic bugs. At the time of writing, all bugs have been confirmed, and 34 of them have been fixed with 11 CVEs assigned. *The SQLite developers responded that many of the bugs we reported have been in the code for many years and no fuzzers have ever run across it, despite SQLite being heavily tested (Anon, 2022e) and used in literally millions of applications.*

We manually investigate the bug-triggering queries and find that 31 out of 54 bugs are first detected in non-optimal query plans. Our observation is that some query plans are rarely executed by DBMS and therefore are less tested. MPE can execute these less-tested query plans to find more bugs, demonstrating the effectiveness of MPE.

Another noteworthy observation is that Kangaroo detects more crash-related bugs than logic bugs, which aligns with the findings of previous studies on logic bug detection (Rigger and Su, 2020a,b,c). This is primarily because crash-related bugs are generally easier to detect, for two key reasons. First, logic bug detection typically relies on specialized oracles that are inherently limited in scope. For instance, our prototype employs MPE, which is designed to detect logic bugs in SELECT queries involving multiple query plans. In contrast, crash bugs can be detected across a wider range of SQL statements, including INSERT, CREATE, and others, thereby increasing the likelihood of their discovery. Second, detecting logic bugs requires high-quality inputs that are both semantically valid and capable of producing comparable results. Crash bugs, on the other hand, may be triggered even by invalid inputs, making them more likely to be uncovered during fuzzing.

5.2. Generating valid queries

As aforementioned, query generation consists of two phases: query structure generation and variable instantiation. The difficulty of variable instantiation can vary substantially across different SQL statements. To fairly evaluate the effectiveness of different variable instantiation approaches, we construct a benchmark that consists of a set of predefined SQL skeletons (i.e., SQL statements with all variables stripped). Each approach is evaluated by instantiating these skeletons and verifying the validity of the resulting statements through execution on the target DBMS. To ensure broad coverage of SQL grammar features, we extract source SQL statements from the official unit tests of the target DBMSs. The benchmark is then constructed by replacing all variables in the source statements with representative placeholders (e.g., "v" for identifiers and 1 for integer constants), according to their respective data types.

We compare the performance of generating valid queries across Kangaroo, Squirrel+, and SQLRight. Both Squirrel+ and SQLRight employ type-sensitive instantiation to concretize variables, whereas Kangaroo adopts a context-sensitive instantiation strategy. SQLancer is also a well-known DBMS testing tool. However, it primarily focuses on oracle design rather than input generation. Therefore, its generator is tailored to produce queries that conform to the specific models required by its oracles, rather than to maximize input diversity and semantic validity. Furthermore, SQLancer adopts a fundamentally different generation strategy which incrementally builds queries clause by clause and assigning values during the generation of each clause. As a result, SQLancer is not capable of instantiating variables within predefined SQL skeletons, making it incompatible with this evaluation setup. Additionally, SQLancer supports only a limited subset of SQL grammar, making it easier to generate valid queries but limiting their variety. Owing to these differences, it is not feasible to conduct a fair standalone comparison of SQLancer's variable instantiation strategy. Following previous works (Liang et al., 2022; Tang et al., 2023), we treat SQLancer's query generator as a whole component and include it only in the overall comparison presented in Section 5.3.

Table 5 shows our evaluation results. Compared to SQLRight and Squirrel+, Kangaroo achieves the highest semantic correctness in all three DBMSs. Since both Kangaroo and SQLRight support full SQL grammars, the results indicate that context-sensitive instantiation outperforms previous work due to richer and more accurate semantic constraints rather than richer SQL grammars supported. SQLRight performs slightly better than Squirrel+ because it considers extra inter-statement constraints.

Another interesting observation is that all these tools achieve significantly better results in SQLite compare to other DBMSs. The primary reason lies in the differing levels of semantic strictness enforced by

Table 4
Real-world Bugs Newly Detected by Kangaroo.

ID	Type	Function	Status	Severity	Reference
SQLite					
1	CS	fts3 and snippet	Fixed	–	cve-2020-23568
2	CS	fts3 and matchinfo	Fixed	–	cve-2020-23569
3	CS	fts3 and ALTER	Fixed	–	cve-2020-23570
4	CS	multi-or covering index	Fixed	–	376e07
5	CS	LIKE and OR optimizer	Fixed	–	84fe52
6	CS	UNION ALL	Fixed	–	2aa354
7	LB	equivalence transfer	Fixed	–	13976a
8	LB	IS NOT NULL AND expr	Fixed	–	6a1424
9	LB	GROUP BY NULL	Fixed	–	0094d8
10	LB	type affinity	Fixed	–	0c437a
11	LB	expression tree	Fixed	–	6a1424
12	AF	query flatten	Fixed	–	a97bbd
13	AF	aggregate queries	Fixed	–	d49628
14	AF	NEVER()	Fixed	–	bfb7ce
15	AF	window function	Fixed	–	9d5aa9
PostgreSQL					
1	CS	empty column value	Fixed	–	#17477
2	AF	table alias	Fixed	–	#17480
MySQL					
1	CS	parser	Fixed	S1	cve-2021-2427
2	CS	parser	Fixed	S1	cve-2022-21303
3	CS	storage	Fixed	S1	cve-2022-21304
4	CS	val_int	Fixed	S1	cve-2022-21640
5	CS	LEX	Fixed	S1	cve-2022-39400
6	CS	find_item	Fixed	S1	cve-2022-21638
7	CS	fix_semijoin_strategies	Fixed	S1	cve-2022-21638
8	CS	make_active_options	Fixed	S1	cve-2022-39400
9	CS	table_contextualize	Fixed	S1	cve-2022-21528
10	CS	query_block_is_recursive	Fixed	S2	cve-2023-21917[a]
11	CS	create_tmp_table	Fixed	S1	107825
12	CS	WITH RECURSIVE	Confirmed	S1	S1649226
13	LB	materialization_lookup	Confirmed	S3	107576
14	LB	semi_and_left_join	Confirmed	S3	107585
15	LB	indexed_materialization	Confirmed	S3	107629
16	LB	UNION	Confirmed	S3	S1651202
17	LB	index	Confirmed	S3	112802
18	LB	hash_join	Confirmed	S2	112816
19	LB	cast	Confirmed	S2	112910
20	LB	primary_key_and_dupsweedout	Confirmed	S2	112911
21	AF	CREATE VIEW UNION	Fixed	S6	107471
22	AF	cond_bool_func	Fixed	S6	107578
23	AF	val_real	Fixed	S6	107638
24	AF	join_read_const_table	Fixed	S6	107681
25	AF	ft_init_boolean_search	Fixed	S6	107733
26	AF	tmp_table_field_type	Fixed	S6	107826
27	AF	optimize_aggregated_query	Fixed	S6	107647
28	AF	ha_index_init	Confirmed	S6	107636
29	AF	val_decimal	Confirmed	S6	107660
30	AF	select_in_like_transformer	Confirmed	S6	107661
31	AF	create_ref_for_key	Confirmed	S6	107663
32	AF	recalculate_lateral_deps	Confirmed	S6	107704
33	AF	fix_outer_field	Confirmed	S6	107719
34	AF	join_read_key_unlock_row	Confirmed	S6	107722
35	AF	having_as_tmp_table_cond	Confirmed	S6	107723
36	AF	add_key_field	Confirmed	S6	107768
37	AF	mdl_request_init	Confirmed	S6	108237

LB: Logic Bugs CS: Crashes AF: Assertion Failure
S1: Critical S2: Serious S3:Non-critical S6:Debug builds

[a] All bugs were submitted to DBMS developers between 2020 and 2022. MySQL developers assign CVE numbers only after the bugs are fixed, which may take over a year.

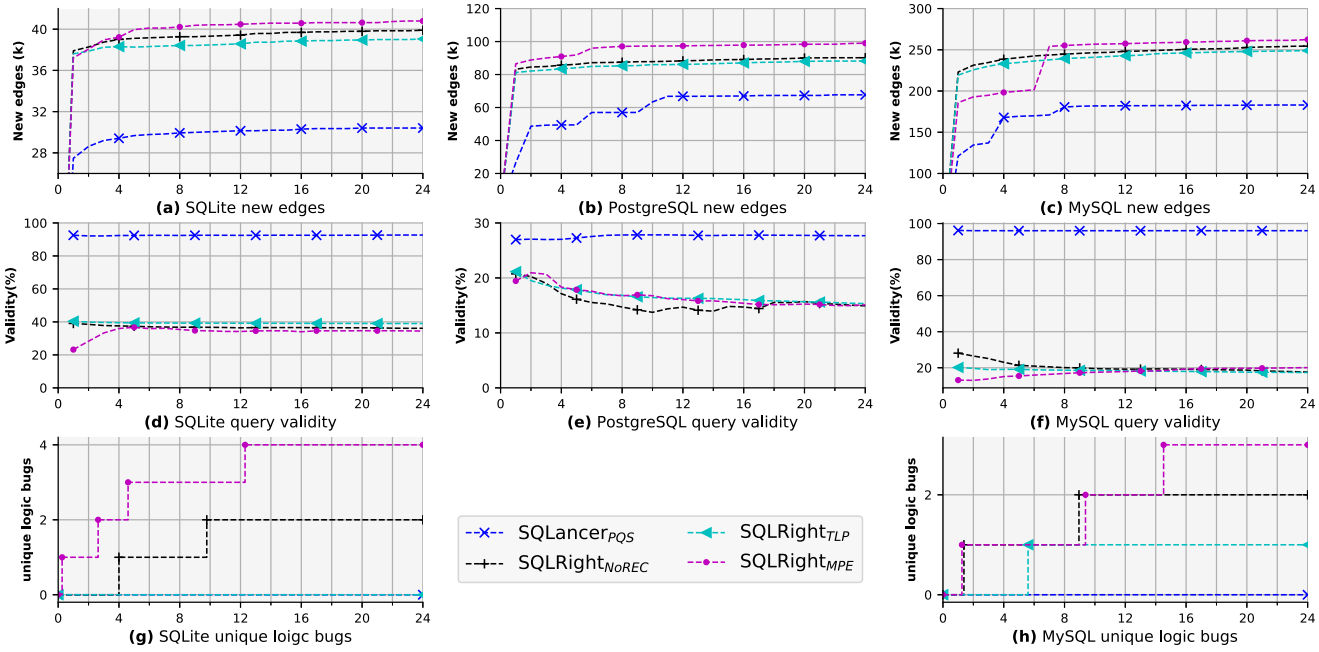

Fig. 6. Comparison between different oracles. It shows the number of code coverage, query validity, and unique logic bugs over time. We run each fuzzing instance for 24 h, repeat each fuzzing for five times. We exclude the results of detected bugs in PostgreSQL as none of the fuzzing instances find any bugs within 24 h.

Table 5
The percentage of semantic correctness queries.

	SQLite	PostgreSQL	MySQL
Squirrel+	24,792(60.1%)	5869(24.9%)	16,283(31.1%)
SQLRight	28,710(61.5%)	8508(26.0%)	35,012(32.5%)
Kangaroo	32,880(70.4%)	18,569(56.7%)	49,130(45.6%)
Total	46,672	32,753	107,693

these DBMSs. SQLite, aligning with Postel's Law (Postel et al., 1981), enforces significantly more relaxed semantic constraints compared to PostgreSQL and MySQL. For example, the query "SELECT 'a' + 1"; is accepted in SQLite but is rejected by PostgreSQL. Consequently, SQLite requires fewer semantic constraints for the same SQL statements and thereby exhibits a higher likelihood of generating valid test cases. Due to the same reason, Kangaroo achieves a smaller improvement on SQLite than other DBMSs.

Kangaroo cannot achieve full semantic validity for two reasons. First, context-sensitive instantiation does not consider dynamic constraints. Second, the static constraints are incomplete since our prototype only implements analysis for primary patterns of major SQL clauses (Section 4.6).

5.3. Comparisons with other techniques

Effectiveness of MPE. To ensure a fair comparison between the logic test oracles, we port MPE to SQLRight to build SQLRight$_{MPE}$ and compare it to SQLRight using different logic test oracles, including NoREC, TLP, and MPE. Doing a fair comparison between MPE and PQS is difficult. PQS requires a generation method to produce the ground-truth result for each SELECT statement during test case generation, making it incompatible with mutation-based fuzzers. Additionally, it demands significant implementation effort and introduces considerable runtime overhead in test case generation. Considering that the support test case generation method is one of the evaluation metrics, we directly use SQLancer$_{PQS}$ as the comparison target. In summary, the effectiveness of logic test oracles is assessed by comparing SQLRight with various oracles and SQLancer$_{PQS}$. As aforementioned, we perform the evaluation on each fuzzer for 24 h, repeat for 5 times.

As shown in Fig. 6, SQLRight$_{MPE}$ outperforms other oracles on all three DBMSs. Specifically, MPE covers slightly more edges (2.24%–12.27%) than other test oracles also used by SQLRight. More importantly, compared to PQS, NoREC, and TLP, MPE finds 7, 3, and 5 more logic bugs, respectively. The main reason for MPE's improvement in bug detection and edge coverage is that it does not put any limitations on queries within test cases. This enables MPE to test a broader range of DBMS functionalities.

Table 6 presents the distribution of detected logic bugs, where all logic bugs found by NoREC are covered by MPE. This demonstrates that MPE can cover the query plans used by queries transformed by NoREC, enabling it to detect more bugs. MPE does not find the only logic bug detected by TLP. This is because TLP is not an oracle that compares different query plans for the same query. TLP transforms the query into a union of three queries and compares the results before and after the transformation. The query plan for the transformed query differs from all query plans of the original query.

Effectiveness of Context-sensitive Instantiation. The comparison between Kangaroo$_{MPE}$, SQLRight$_{MPE}$, and SQLancer$_{MPE}$ shows the overall effectiveness of their query generators. As expected, SQLancer$_{MPE}$ achieves the highest query validity. This is primarily due to its adherence to a simplified version of SQL grammar. However, this design also constrains the diversity and complexity of the generated queries. For instance, SQLancer does not support the generation of subqueries, which allows it to bypass the complex constraints introduced by multi-level nested subqueries. That makes it easier to generate valid statements but limits the diversity. That explains why it explores the fewest paths among these three fuzzing systems. In contrast, both Kangaroo$_{MPE}$ and SQLRight$_{MPE}$ produce more diverse queries, leading to better path coverage and bug detection. Benefiting from the richer semantic constraints, Kangaroo$_{MPE}$ achieves significantly higher query validity than SQLRight$_{MPE}$.

To evaluate the contribution of context-sensitive instantiation to logic bug detection, we compare Kangaroo$_{MPE}$ to SQLRight$_{MPE}$ because the only difference between these two fuzzers is the variable instantiation strategy used. As shown in Fig. 7, Kangaroo$_{MPE}$ outperforms SQLRight$_{MPE}$ on all three DBMSs. The results confirm that context-sensitive instantiation can significantly improve the performance of logic bug detection compared to type-sensitive instantiation.

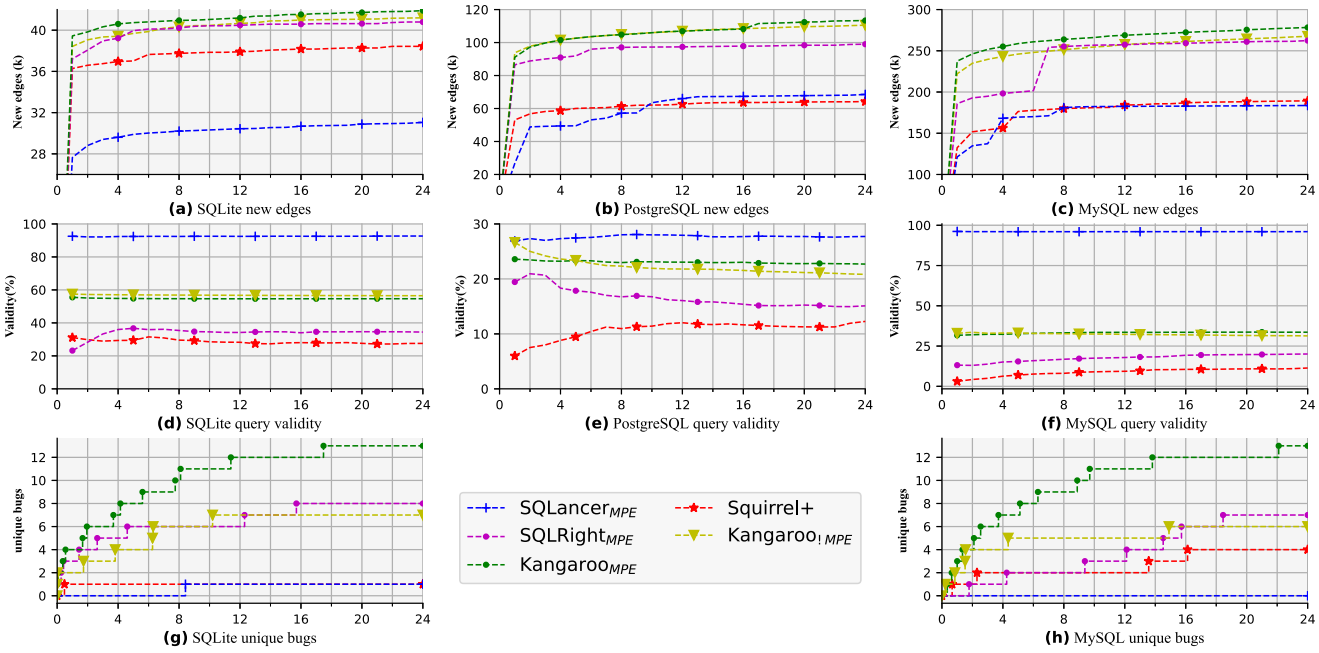

Fig. 7. Comparison between different query generators. It illustrates the contributions of context-sensitive instantiation for bug detection. Table 7 presents the distribution of logic bugs and crash-related bugs. We exclude the results of detected bugs in PostgreSQL as only Kangaroo$_{MPE}$ found one crash.

Table 6
Distribution of logic bugs found in DBMS within 24 h by different oracles.

ID	DBMS	MPE	NoREC	TLP	PQS
1	SQLite	✓	×	×	×
2	SQLite	✓	✓	×	×
3	SQLite	✓	×	×	×
4	SQLite	✓	✓	×	×
5	MySQL	✓	✓	×	×
6	MySQL	✓	✓	×	×
7	MySQL	✓	×	×	×
8	MySQL	×	×	✓	×

Table 7
Number of bugs detected in three DBMSs within 24 h. Value in brackets indicates the number of bugs detected in SQLite, PostgreSQL, and MySQL, respectively.

Fuzzer	Crash-related bugs	Logic bugs	Total
SQLRight$_{MPE}$	8(4/0/4)	7(4/0/3)	15(8/0/7)
Kangaroo$_{MPE}$	16(8/1/7)	11(5/0/6)	27(13/1/13)
Squirrel+	5(1/0/4)	–	5(1/0/4)
Kangaroo$_{!MPE}$	13(7/0/6)	–	13(7/0/6)

Specifically, it achieves around 55%, 23%, and 33% validity for three tested DBMSs, which is significantly higher than SQLRight$_{MPE}$. This proves that context-sensitive instantiation is helpful in improving query validity. Due to the higher query validity, Kangaroo$_{MPE}$ wastes less time in testing invalid queries and explores more edges, thereby finding more logic bugs than SQLRight$_{MPE}$. Specifically, compared to SQLRight$_{MPE}$, Kangaroo$_{MPE}$ explores 14% more edges on average and discovers 4 more logic bugs in total.

To evaluate the contribution of context-sensitive instantiation to crash-related bug detection, we disable the MPE in Kangaroo to build Kangaroo$_{!MPE}$ and compare it to Squirrel+. Without logic bug test oracles, these two fuzzer settings focus on crash-related bug detection and utilize different variable instantiation strategies. Because Kangaroo$_{!MPE}$ has a higher probability of generating valid queries, it can better explore the deep logic of DBMS and find more bugs. Specifically, Kangaroo$_{!MPE}$ explores 39.91% more edges on average and finds 8 more bugs than Squirrel+ with the help of context-sensitive instantiation. These results confirm the effectiveness of context-sensitive instantiation in bug discovery.

5.4. Benefits of the two key techniques

We conduct an ablation study to evaluate the individual contributions of the two key techniques. Fig. 7 shows the evaluation results.

Context-Sensitive Instantiation As shown in Fig. 7def, tools that employ *context-sensitive instantiation* achieve significantly higher semantic correctness than other tools. Consequently, it significantly improves the discovery of edges and crash-related bugs (from 5 to 13). More importantly, Kangaroo$_{MPE}$ discovers 5 additional logic bugs that were missed by SQLRight$_{MPE}$ with the help of context-sensitive instantiation.. This result confirms that query validity plays an important role in effectively detecting logic bugs. This is reasonable because an invalid query cannot return results used for logic bug checking. In conclusion, context-sensitive instantiation can significantly improve the efficiency of bug detection, especially for logic bugs, by enhancing the validity of generated test cases.

Multi-Plan Execution The comparison between Kangaroo and Kangaroo$_{!MPE}$ reveals the effect of MPE. First, MPE enables logic bug detection in SELECT statements that contain multiple query plans. Such statements are prevalent in practice. In our experiments, the SELECT statements generated by the system produce an average of 12.3, 15.7, and 16.1 query plans in SQLite, PostgreSQL, and MySQL, respectively. Interestingly, MPE also shows a slight improvement in detecting crash-related bugs. Kangaroo captures four more crash-related bugs than Kangaroo$_{!MPE}$ in total, which is attributed to the exploration of rarely executed query plans.

6. Discussion and limitations

6.1. Limitations of context-sensitive instantiation

Our context-sensitive instantiation strategy is theoretically capable of inferring both complete and sound semantic constraints. However, similar to existing work (Zhong et al., 2020; Liang et al., 2022, 2024), we currently enforce only static constraints. This restriction reduces

the validity of generated queries involving dynamic behavior. This limitation arises from the practical challenges associated with dynamic constraints, which significantly increase the complexity of constraint reasoning. Furthermore, enforcing dynamic constraints would introduce considerable overhead during test case generation, potentially lowering system efficiency. Thus, we restrict our enforcement to static constraints, which capture the majority of semantically invalid cases while maintaining efficiency. We consider supporting dynamic constraint enforcement as part of future work.

The theoretical completeness and soundness of context-sensitive constraint inference depend on the precise fine-grained semantic analysis of SQL clauses. In practice, there are two main sources that may lead to constraint inaccuracies: (1) implementation errors in the semantic analysis for certain SQL clauses, and (2) lack of fine-grained constraint analysis for some clauses due to limited engineering effort. We do not view these issues as essential limitations of the approach itself. Instead, they reflect practical limitations of the current prototype and can be addressed through further development.

6.2. Limitations of MPE

Similar to existing oracles for detecting logic bugs in DBMSs, MPE also suffers from false positives (FP) and false negatives (FN). Below, we discuss these limitations and the strategies we employ to mitigate them in practice.

False Negatives Due to Shared Incorrect Results MPE may fail to detect certain logic bugs (i.e., false negatives) when all query plans for a given SQL statement yield the same incorrect result. This limitation is inherent to oracles that rely on differential behavior across alternative execution paths, such as PQS, TLP, and TQS. However, in comparison to these oracles, MPE reduces the likelihood of such false negatives by exploring a larger set of query plans. Furthermore, bugs that manifest consistently across all query plans typically stem from low-level components. These components are often well covered by existing unit testing frameworks (Anon, 2023b).

False Positives from Non-Deterministic Queries SQL queries can be ambiguous due to the non-deterministic behavior in DBMSs. Therefore, inconsistencies between the results of different query plans for non-deterministic queries do not necessarily indicate bugs. Such queries are common and can significantly reduce the effectiveness of test oracles, which also pose challenges to previous works (Rigger and Su, 2020b,a; Liang et al., 2022, 2024). To better understand and mitigate this issue, we evaluate a wide range of features in each tested DBMS. Our evaluation identifies four primary sources of non-determinism.

The first source of non-determinism comes from undefined data access order. In most cases, it only impacts the row order in results, which can be eliminated by sorting the results. However, the non-determinism cannot be resolved if the statement contains certain SQL features. For instance, the `LIMIT` clause results in only part of the records being fetched. In such cases, the final output may vary, depending on the data access order during query execution. The second source is the non-deterministic functions or variables such as `random()` and `CURRENT_TIME`. The third source is the loss of precision, which is common in DBMSs' built-in statistical analysis functions. Finally, some non-deterministic behaviors depend on the dynamic execution context. For example, SQLite assigns an affinity type to each column and uses different comparison algorithms for different affinity types. The statement `CREATE VIEW v1(c1) AS SELECT c0 FROM t1 UNION SELECT c0=10 FROM t1;` involves two SELECT statements. SQLite is free to choose the execution order of the two `SELECT` statements, and the affinity type of column `c1` depends on this order. This can lead to inconsistent results in subsequent queries, such as `SELECT * FROM v1 NATURAL JOIN v1;`.

To mitigate false positives caused by such non-determinism, we perform feature-aware analysis during context-sensitive instantiation. Specifically, we identify queries containing non-deterministic features (e.g., the `LIMIT` clause), and exclude them from result comparison. Importantly, this strategy does not compromise our ability to detect memory-related bugs in these queries, as it only omits the result comparison step.

False Positives from Non-Triggerable Plans MPE requires modifying DBMSs, which might generate non-triggerable query plans and cause false positives. However, this rarely occurs since our intervention is limited to selecting among query plans generated by the DBMS, without altering the plan generation process. In practice, we encountered only one such instance, where a query plan **consistently** incurred higher costs compared to an alternative query plan. Specifically, the query plan attempted to perform a parallel table scan with only a single worker, which proved to be less efficient than a non-parallel scan. Consequently, this query plan would never be chosen for execution by the original DBMS. However, this case can also be viewed as a performance bug since the DBMS wastes efforts on generating meaningless query plans.

6.3. Threats to validity

External validity A potential external validity threat is that the DBMSs selected for our experiment are all C/C++-based. This is because our system prototype is built on AFL, which can only statically instrument C/C++ programs to collect coverage information. However, the two techniques we propose are not limited by this constraint. Both of them can be applied to other fuzzing tools to detect bugs in non-C/C++-based DBMS.

Internal validity Minimizing systematic errors in the evaluation process is essential. Since fuzzing involves inherent randomness, we mitigate this threat by repeating each experiment five times and reporting the average results. To ensure consistent and comparable coverage measurements, we evaluate the coverage achieved by all fuzzers on the same instrumented binary To avoid selection bias, we utilize the same set of initial seeds for all fuzzers, except SQLancer, which does not require any seeds.

Conclusion validity Evaluation metrics may introduce potential threats to the results. In this paper, we selected commonly used, comprehensive indicators in previous bug detection works. To isolate the impact of the technique, we applied a controlled variable strategy in the comparison experiment. For example, when evaluating the effectiveness of different test oracles, we applied the same test tools to all oracles. The only exception is PQS, because it inherently requires a distinct test case generation method.

6.4. Practical applicability and integration

PoC Generation A test case that triggers a bug on the modified DBMS may fail to reproduce the bug on the original DBMS if the issue arises only with a non-optimal query plan. Such a test case cannot directly serve as proof-of-concept (PoC) (Anon, 2022i) that demonstrates the bug on the original system. To save DBMS developers' effort, we submit DBMS diagnostic logs along with corresponding minimal PoCs

To build a minimal PoC, we first try to remove statements or clauses in the test case and check whether the bug is still triggered. Then, we manually adjust the minimal test cases to trigger bugs on the unmodified DBMS. Further details on the manual adjustment are provided in Appendix A.3. In practice, we spent about 30 h manually constructing PoCs for all 31 bugs detected in non-optimal query plans.

Effort of Adoption Applying Kangaroo to a new DBMS requires domain knowledge of its grammar and query plan selection implementation. This requirement, however, aligns well with the expertise of the primary users of DBMS testing tools, i.e., DBMS developers. Developers can adopt Kangaroo by following the steps below:

Table 8
Effort of adoption to DBMSs.

Component(LOC)	SQLite	PostgreSQL	MySQL
Semantic configuration	102	109	132
Parser	52	161	63
Context-sensitive instantiation	106	135	259
DBMS modification	248	271	191
Total Person-hours	32	40	45

(1) *Define semantic mappings*: Define the mapping rules in the semantic configuration to help identify the alias of standard SQL clauses.
(2) *Augment syntax validation*: Some DBMSs perform additional syntax checks beyond BNF rules, which cannot be automatically ported. These checks must be manually ported to improve the accuracy of generated parser.
(3) *Support DBMS-specific extensions*: Add semantic definitions and constraint collection logic for DBMS-specific clauses to ensure proper support for non-standard SQL features.
(4) *Refine constraint collection*: For clause patterns whose semantic requirements deviate from general, override the default constraint collection logic.
(5) *Enable multi-plan execution*: Modify the DBMS to support the execution of all valid query plans.

Table 8 summarizes the human effort of adopting our prototype to specific DBMSs.

Extend to other statements MPE can also be applied to other statements with multiple execution plans, such as INSERT and UPDATE. This extension introduces two main challenges. First, non-SELECT statements do not yield directly comparable results since they typically do not return detailed information about data modifications. To address this, a follow-up query can be issued after executing each plan to retrieve information about the affected database objects (e.g., tables). Second, executing one query plan may modify the database state, which can affect the execution of subsequent plans. This can be addressed by using the DBMS transaction rollback mechanism to restore the original state before executing the following plans.

Integration into testing pipelines Recently, several mainstream DBMSs have adopted fuzzing as an integral part of their official testing frameworks. For example, SQLite integrated fuzzing into its internal test suite in 2015 as a complementary approach to unit testing. Since Kangaroo is also a fuzzing-based tool, it can be seamlessly integrated into existing official DBMS fuzzing workflows.

Fuzzing is generally not well-suited for integration into main Continuous Integration (CI) pipelines due to its non-deterministic behavior, substantial computational overhead, and long execution times. Instead, it is typically employed as a complementary testing technique to uncover vulnerabilities that may be missed by conventional methods. In practice, fuzzing is often executed as a separate nightly or scheduled job. Although these jobs fall outside the core CI pipeline, they constitute an essential part of the broader testing framework. This asynchronous execution enables developers to leverage the fuzzing's strengths in exploring execution paths without impacting the performance or responsiveness of the CI pipeline. Moreover, these jobs can be integrated within the Continuous Delivery (CD) process to provide deeper verification of software robustness and security before release. Listing 2 in the Appendix illustrates a sample configuration for running nightly fuzzing jobs with GitHub Actions.

By default, our system interacts with the target DBMS through its native API (e.g., libpq, libmysqlclient) to ensure efficient query execution and result retrieval. It also supports communication via standardized interfaces such as Open Database Connectivity (ODBC) and command-line interfaces (CLI). This interface flexibility facilitates integration of Kangaroo into existing testing pipelines.

7. Related work

Oracles for detecting DBMS semantic bugs. DBMS semantic bug detection relies on test oracles to identify unexpected behavior, such as performance regressions and incorrect results. We can classify semantic bug detection approaches into three categories.

The first approach is based on differential testing which executes a given input with different DBMSs. Slutz proposed RAGE (Slutz, 1998) for finding logic bugs by running the same queries on different DBMSs and comparing their results. Jinho et al. developed APOLLO (Jung et al., 2019) to find performance regression bugs by executing the same query on the DBMSs with different versions. However, RAGE can only be applied to common features of different DBMSs, and APOLLO can only detect bugs introduced or fixed by newer versions.

Another approach is based on metamorphic testing which identifies bugs by running two queries with known relationships between their results. If their results do not conform to the expected relationships, a potential logic bug is detected. However, all of them (Sutton et al., 2007; Tang et al., 2023; Rigger and Su, 2020a,b,c; Ba and Rigger, 2024) are limited to the queries that can be converted.

The last approach tries to build the test case along with the corresponding ground-truth result. ADUSA (Khalek et al., 2008) generates all data and the full expected result for a query. However, generating full expected results can be expensive which inhibits it from finding more bugs. To simplify the ground truth generation, Pivoted Query Synthesis(PQS) (Rigger and Su, 2020c) only partly validates a query's result. It synthesizes a query expected to fetch a single, randomly-selected row and detects logic bugs by checking whether this row is fetched. Similar to NoREC, PQS is also mostly limited to finding bugs in `WHERE` clauses.

DBMS test cases generation. DBMS requires structural inputs to manipulate data in the database. Structural input generation mainly falls into two categories: generate-based approaches and mutation-based approaches.

The generation-based approach (Rigger and Su, 2020a,c,b; Slutz, 1998; Andreas Seltenreich, 2022; Binnig et al., 2007; Khalek et al., 2008; Tang et al., 2023) is effective in generating syntax-correct test cases since it typically follows a grammar model. However, these grammar rules are helpless in improving the semantic correctness of the test case. For example, SQLsmith (Andreas Seltenreich, 2022) can generate syntax-correct test cases from abstract syntax trees. It achieves quite a low accuracy on semantics which might inhibit it from finding bugs hidden in the deep logic. QAGen (Binnig et al., 2007) proves that generating a completely valid query is NP-complete. It improves semantic correctness by combining traditional query processing and symbolic execution. Previous works also try to improve query generation by generating queries that satisfy certain constraints (Khalek et al., 2008). They reduce the query generation into the SAT problem, which is subsequently solved by a solver (e.g., Alloy (Anon, 2022b)).

The mutation-based approach incorporates execution feedback to explore the deep logic of tested programs. The general fuzzers (Stephens et al., 2016; Yun et al., 2018; Chen and Chen, 2018; Chen et al., 2020; Gan et al., 2020) unaware of the input structure. They can hardly reach the deep logic of DBMSs even incorporating advanced techniques such as taint analysis or symbolic execution. Blazytko et al. (2019) utilizes grammar-like combinations to synthesize highly structured inputs without the need for explicit grammar, but most of the generated test cases are still syntax invalid. Recent works manually provide grammar specifications to guide mutation as they guarantee that the generated queries have correct grammar. For example, Bati et al. (2007) propose a genetic approach to mutate SQL by inserting, replacing, or removing grammar with the guidance of execution feedback such as query results and query plans.

8. Conclusion

This paper presents Kangaroo, a mutation-based system for detecting both logic and crash-related bugs in DBMS. It proposed two key techniques, i.e., context-sensitive instantiation to generate semantically valid SQL queries during mutation and multi-plan execution that can detect logic bugs in the DBMS execution engine. We developed a prototype system and applied it to three widely used DBMSs. It successfully identified 54 unique bugs. The further evaluation shows that Kangaroo outperforms existing tools.

CRediT authorship contribution statement

Jiaqi Li: Writing – review & editing, Writing – original draft, Validation, Supervision, Software, Project administration, Methodology, Conceptualization. **Ke Wang:** Software, Investigation. **Yaoguang Chen:** Software. **Yajin Zhou:** Writing – review & editing, Supervision, Resources, Funding acquisition. **Lei Wu:** Writing – review & editing. **Jiashui Wang:** Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Jiaqi Li, Yajin Zhou, Lei Wu has patent pending to 2025100223417. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments and feedback. This work was supported by the National Key R&D Program of China (No. 2022YFE0113200), the National Natural Science Foundation of China (NSFC) under Grant U21A20464. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

Appendix

```yaml
name: Nightly Fuzzing

on:
  schedule:
    - cron: '0 1 * * *' # Run daily at 1:00 AM UTC
  workflow_dispatch: # Allow manual trigger

jobs:
  fuzz:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Set up dependencies
        run: |
          ......

      - name: Build fuzzer toolchain
        run: |
          cd /path/to/Kangaroo
          make # build fuzzer
          cd AFL
          make # Build compiler for instrumentation

      - name: Build instrumented DBMS
        run: |
          # Specify the compiler for instrumentation
          export CC= /path/to/afl-gcc
          export CXX= /path/to/afl-g++
          # Build instrumented DBMS
          mkdir build
          cd build
          cmake -DENABLE_MPE=ON .. # Assume MPE has been integrated
          make fuzz_target

      - name: Run fuzzing
        run: |
           cd /path/to/fuzz/dir
           # Run the fuzzing script
           # It first sets up the DBMS, and then launches the fuzzer to test it.
          ./run_kangaroo.sh

      - name: Upload crash artifacts (if any)
        if: always()
        uses: actions/upload-artifact@v3
        with:
          name: fuzz-crashes
          path: /path/to/fuzz/dir/output/crash
```

Listing 2 Example Nightly Fuzzing Workflow Configuration for GitHub Actions

A.1. Implementation detail of constraint collection

Kangaroo implements a `GetConstraint` function for each clause pattern. Fig. 8 illustrates the `GetConstraint` functions for several SQL clause patterns. For example, in Fig. 8(a), the `GetConstraint` function for the first pattern in JoinedTabled can infer, based on the context of the current pattern, that the scope of the `TableRef` identifiers, i.e., columns of this joined table can be referenced by `ColumnRef` identifiers within `JoinConstraints`. In Fig. 8(b), the `GetConstraint` functions can capture the attributes of a `ColumnDef` identifier when defined and infer the attribute requirements for identifiers.

The semantic constraints of the same clause pattern may vary across different DBMSs. For example, the pattern `Expression '+' Expression` generally requires both operands to be numeric; however, SQLite permits addition operations between operands of any type. To address this, when a DBMS enforces semantic constraints that differ from the general behavior, we implement a new `GetConstraint` function to override the original one. Furthermore, some DBMSs have their own SQL dialect. As aforementioned, the semantic tree grammar also contains the SQL dialect of the tested DBMS. Therefore, we can implement the `GetConstraint` function for the patterns of these dialects. In addition, we provide a set of APIs to capture semantic information and insert various semantic constraints, significantly simplifying the engineering effort required to implement the `GetConstraint` function. These design choices facilitate the adaptation of our methodology to other DBMSs with minimal engineering effort.

A.2. Randomized backtracking algorithm example

The process of the randomized backtracking algorithm is as follows. First, we obtain a topological order of variables based on their dependencies and instantiate the variables sequentially according to this order. During each instantiation, a value is randomly assigned to the variable such that all constraints imposed on it are satisfied. If no valid solution can be found for a variable, the algorithm backtracks to the variables it depends on and re-instantiate them.

For example, in Fig. 2, `x2`, `x3`, and `x6` only depend on previously declared tables `t0` and `t1`, so we first instantiate them in turn by randomly selecting a referenced table. Next, we instantiate `x1`, `x4`, and `x5`, as all their dependencies have already been instantiated. Suppose we concretize both `x2` and `x3` with table `t1`, the joined table will have two columns with conflicting names. When concretizing identifier `x4`, the column dependency requires that `x4` should be a column with a unique name in the joined table, but no dependencies satisfy this constraint. In this case, we will re-instantiate `x3`, `x4`, and all variables that depend on them.

API to add constraints

- **TypeConstraint():** Assigns the semantic type to identifiers and constants.
- **RelyConstraint():** Establishes dependencies between identifiers.
- **AttrConstraint():** Specifies attribute requirements for identifiers.

SQL clause pattern and corresponding GetConstraint() function

```
SingleTableRef:
| IDENT                // Identifier
  General::GetConstraint() {
    TypeConstraint($1, TableRef);  // $i represents the i-th element in the pattern
    RelyConstraint(TableRefRely, $1, TableInDatabase);
  }
.....
JoinedTable:
| TableReference JOIN TableReference JoinConstraints
  General::GetConstraint() {
    tmpTable = BuildJoinedTable($1, $3);
    // Identifiers in JoinConstraints clause can only refer to columns of tmpTable
    for(ident : GetColumnRefList($4)) {
      RelyConstraint(ColumnRefRely, ident, tmpTable);
    }
  }                    (a) Accurate identifier scoping to obtain precise constraints
.....
ColumnDef:
| ColumnName ColumnType OptFieldAttributes
  General::GetConstraint() {
    TypeConstraint($1, ColumnDef);
    BuildColumn($1, $2, $3); // Set the attributes of column
  }                    (b) Capture attributes of defined identifiers
.....
Expression:
| Expression '+' Expression
  General::GetConstraint() {
    // The data types of $1 and $3 should match the operator $2.
    AttrConstraint(DataTypeMatch, $1, $3, $2);
  }                    (b) Infer attribute restrictions for identifiers
  SQLite::GetConstraint() { // Function overloading for SQLite
    // Do nothing
  }                    (c) Customize constrain collection for different DBMSs
| Expression ILIKE Expression
  PostgreSQL::GetConstraint() {
    AttrConstraint(DataType, $1, TEXT);  // The data type of $1 should be TEXT
    AttrConstraint(DataType, $3, TEXT);
  }                    (d) Infer constraints from DBMS dialect
.....
```

Fig. 8. An example to show some `GetConstraint` functions. We provide a set of APIs to simplify the implementation of `GetConstraint` function. The figure only lists the three APIs used to insert different types of constraints.

A.3. PoC generation strategies

MPE can only conduct the PoC on **modified** DBMSs. To build PoC on **unmodified** DBMSs, our goal is to make the buggy query plan to be optimal. The ways to achieve this goal can be classified into three categories.

First, force a query plan to be chosen by using a hint to give DBMS an explicit optimization instruction. For example, in MySQL, the hint FORCE INDEX forces a table scan to use the specified index if possible. If the buggy query plan scans a table with an index and others do not, this hint could force the plan to be chosen.

Second, eliminate more efficient query plans. That can be achieved by turning off certain optimizations, which is feasible since DBMSs typically provide a configuration to switch on/off some optimization strategies. Sometimes, using hints (e.g. IGNORE INDEX) can also eliminate some query plans. Besides, rewriting test cases can achieve this goal as well. For example, by removing the CREATE INDEX statement, query plans that use this index will be eliminated.

Third, modify the cost of query plans by adjusting either *action count* or *action cost*. A query plan is composed of a series of operations, and the cost of an operation can be simplified as *action count* * *action cost*. *action count* is the amount of data to process/access, and *action cost* is the overhead of processing one unit of data. In some DBMS, e.g. PostgreSQL, the *action cost* is a serial of parameters that can be set by the user. To change the *action count*, we can modify expressions in the statement, change the DBMS status, or provide hints for SELECT statements. For example, we found a buggy query plan that sequence scans a table perform worse than another use index search performed better because the SELECT statement had a condition filter "$a < 10$". Suppose the table has 100 rows and only 10 of the rows satisfy this condition, the index search will perform better since it requires less data access. If we set the condition to "$a < 100$" such that all rows satisfy this condition, the sequence scan will perform better because the index search operation requires the same *action count* as the sequence scan operation but extra *action cost* to search the index. Modifying the data in the query table to make more data satisfy the expression can achieve the same purpose. Some DBMS allow us to use hints to intervene in the probability that a logical expression is true. For example, we can use hints "$likely(a < 10)$" to tell SQLite that this condition is likely to be true to increase the *action count* of the index search operation.

Data availability

Data Availability. The artifact associated with this article, including experimental data, scripts, and instructions is available at: https://github.com/anonymous44117/Kangaroo.

References

Andreas Seltenreich, S.M., 2022. SQLsmith. https://github.com/anse1/sqlsmith/.

Anon, 2022a. AddressSanitizer. https://github.com/google/sanitizers/.

Anon, 2022b. Alloy. https://alloytools.org/.

Anon, 2022c. American Fuzzy Lop (2.56b). https://lcamtuf.coredump.cx/afl/.

Anon, 2022d. Common vulnerabilities and exposures. https://en.wikipedia.org/wiki/Common_Vulnerabilities_and_Exposures.

Anon, 2022e. How SQLite is tested. https://www.sqlite.org/testing.html.

Anon, 2022f. MySQL customers. https://www.mysql.com/customers/.

Anon, 2022g. MySQL homepage. https://www.mysql.com/.

Anon, 2022h. PostgreSQL homepage. https://www.postgresql.org/.

Anon, 2022i. Proof of concept. https://en.wikipedia.org/wiki/Proof_of_concept#Security.

Anon, 2022j. SQLite homepage. https://www.sqlite.org/.

Anon, 2022k. Well-known users of SQLite. https://www.sqlite.org/famous.html.

Anon, 2023a. Randomized depth-first search. https://en.wikipedia.org/wiki/Maze_generation_algorithm.

Anon, 2023b. Unit testing. https://en.wikipedia.org/wiki/Unit_testing.

Ba, J., Rigger, M., 2024. Keep it simple: Testing databases via differential query plans. Proc. the ACM Manag. Data 2 (3), 1–26.

Bannister, A., 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.

Bati, H., Giakoumakis, L., Herbert, S., Surna, A., 2007. A genetic approach for random testing of database systems. In: Proceedings of the 33rd International Conference on Very Large Data Bases. pp. 1243–1251.

Binnig, C., Kossmann, D., Lo, E., Özsu, M.T., 2007. QAGen: generating query-aware test databases. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. pp. 341–352.

Blazytko, T., Bishop, M., Aschermann, C., Cappos, J., Schlögel, M., Korshun, N., Abbasi, A., Schweighauser, M., Schinzel, S., Schumilo, S., et al., 2019. {GrimoiRE}: Synthesizing structure while fuzzing. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1985–2002.

Chamberlin, D.D., Boyce, R.F., 1974. SEQUEL: A structured english query language. In: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control. pp. 249–264.

Chen, P., Chen, H., 2018. Angora: Efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 711–725.

Chen, Y., Li, P., Xu, J., Guo, S., Zhou, R., Zhang, Y., Wei, T., Lu, L., 2020. Savior: Towards bug-driven hybrid testing. In: 2020 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 1580–1596.

Chen, Y., Zhong, R., Hu, H., Zhang, H., Yang, Y., Wu, D., Lee, W., 2021. One engine to fuzz'em all: Generic language processor testing with semantic validation. In: 2021 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 642–658.

Cimpanu, C., 2019. Google Chrome impacted by new Magellan 2.0 vulnerabilities. https://www.zdnet.com/article/google-chrome-impacted-by-new-magellan-2-0-vulnerabilities/.

Dutra, R., Bachrach, J., Sen, K., 2018. SMTSampler: Efficient stimulus generation from complex SMT constraints. In: 2018 IEEE/ACM International Conference on Computer-Aided Design. ICCAD, IEEE, pp. 1–8.

Fu, J., Liang, J., Wu, Z., Wang, M., Jiang, Y., 2022. Griffin: Grammar-free DBMS fuzzing. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. pp. 1–12.

Gan, S., Zhang, C., Chen, P., Zhao, B., Qin, X., Wu, D., Chen, Z., 2020. {GreyoNE}: Data flow sensitive fuzzing. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2577–2594.

Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., Chen, Z., 2018. Collafl: Path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 679–696.

Ghit, B., Poggi, N., Rosen, J., Xin, R., Boncz, P., 2020. SparkFuzz: Searching correctness regressions in modern query engines. In: Proceedings of the Workshop on Testing Database Systems. pp. 1–6.

Hao, Z., Huang, Q., Wang, C., Wang, J., Zhang, Y., Wu, R., Zhang, C., 2023. Pinolo: Detecting logical bugs in database management systems with approximate query synthesis. In: 2023 USENIX Annual Technical Conference (USENIX ATC 23). pp. 345–358.

Jiang, Z.-M., Bai, J.-J., Su, Z., 2023. {DynsQL}: Stateful fuzzing for database management systems with complex and valid {sQL} query generation. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 4949–4965.

Jung, J., Hu, H., Arulraj, J., Kim, T., Kang, W., 2019. APOLLO: Automatic detection and diagnosis of performance regressions in database systems. Proc. the VLDB Endow. 13 (1), 57–70.

Khalek, S.A., Elkarablieh, B., Laleye, Y.O., Khurshid, S., 2008. Query-aware test generation using a relational constraint solver. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. IEEE, pp. 238–247.

Knuth, D.E., 1964. Backus normal form vs. backus naur form. Commun. ACM 7 (12), 735–736.

Liang, J., Chen, Y., Wu, Z., Fu, J., Wang, M., Jiang, Y., Huang, X., Chen, T., Wang, J., Li, J., 2023. Sequence-oriented DBMS fuzzing. In: 2023 IEEE 39th International Conference on Data Engineering. ICDE, IEEE, pp. 668–681.

Liang, Y., Liu, S., Hu, H., 2022. Detecting logical bugs of {dbMS} with coverage-based guidance. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 4309–4326.

Liang, J., Wu, Z., Fu, J., Wang, M., Sun, C., Jiang, Y., 2024. Mozi: Discovering DBMS bugs via configuration-based equivalent transformation. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 1–12.

Liu, X., Zhou, Q., Arulraj, J., Orso, A., 2021. Automated performance bug detection in database systems. arXiv e-prints arXiv–2105.

Neystadt, J., 2008. Automated penetration testing with white-box fuzzing. MSDN Libr..

Postel, J., et al., 1981. Transmission control protocol. p. 13.

Rigger, M., Su, Z., 2020a. 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. pp. 1140–1152.

Rigger, M., Su, Z., 2020b. Finding bugs in database systems via query partitioning. Proc. the ACM Program. Lang. 4 (OOPSLA), 1–30.

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

Slutz, D.R., 1998. Massive stochastic testing of SQL. In: VLDB, vol. 98, Citeseer, pp. 618–622.

Song, J., Dou, W., Cui, Z., Dai, Q., Wang, W., Wei, J., Zhong, H., Huang, T., 2023. Testing database systems via differential query execution. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 2072–2084.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2016. Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS, vol. 16, (2016), pp. 1–16.

Sutton, M., Greene, A., Amini, P., 2007. Fuzzing: Brute Force Vulnerability Discovery. Pearson Education.

Tang, X., Wu, S., Zhang, D., Li, F., Chen, G., 2023. Detecting logic bugs of join optimizations in DBMS. Proc. the ACM Manag. Data 1 (1), 1–26.

Wang, J., Chen, B., Wei, L., Liu, Y., 2019. Superion: Grammar-aware greybox fuzzing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 724–735.

Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T., 2018. {QsYM}: A practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 745–761.

Zhang, Y., Yao, P., Wu, R., Zhang, C., 2021. Duplicate-sensitivity guided transformation synthesis for DBMS correctness bug detection. arXiv preprint arXiv:2107.03660.

Zhong, R., Chen, Y., Hu, H., Zhang, H., Lee, W., Wu, D., 2020. Squirrel: Testing database management systems with language validity and coverage feedback. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 955–970.