SmartFuzz: Leveraging Large Language Models and Feature Composition to Generate High-Quality Seeds for Database Fuzzing

LI LIN, School of Informatics, Xiamen University, China
JINTAI HONG, School of Informatics, Xiamen University, China
YANLIN ZHUANG, School of Informatics, Xiamen University, China
RONGXIN WU, School of Informatics, Xiamen University, China

Mutation-based fuzzing is one of the most effective techniques for uncovering bugs in Database Management Systems (DBMSs). However, its effectiveness critically depends on the quality of the initial seed queries. High-quality seeds should be syntactically and semantically valid, incorporate diverse SQL features, and encode behaviors that drive execution into bug-prone states. In practice, existing DBMS fuzzers primarily rely on SQL queries extracted from unit tests or regression suites as initial seeds, which are often limited in diversity and scale, leaving many DBMS features and execution paths unexplored.

To address this limitation, we propose SmartFuzz, an automated framework for synthesizing high-quality initial SQL seeds for mutation-based DBMS fuzzing using Large Language Models (LLMs). The key insight behind SmartFuzz is that two underutilized sources—official DBMS documentation and historical crash-triggering inputs—capture complementary knowledge about DBMS feature usage and bug-relevant behaviors. SmartFuzz extracts structured features from these sources and leverages LLMs to synthesize executable, feature-rich SQL seeds that are biased toward bug-prone execution states. We integrate SmartFuzz into existing mutation-based DBMS fuzzing pipelines and evaluate it on 4 widely used DBMSs. The results demonstrate that SmartFuzz significantly improves bug discovery and code coverage compared to state-of-the-art mutation-based fuzzers. In total, SmartFuzz detects 61 previously unknown bugs, of which 60 have been confirmed and fixed by developers.

CCS Concepts: • **Security and privacy → Database and storage security**.

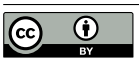
Additional Key Words and Phrases: DBMS Testing, Fuzzing, Seed Generation, Large Language Models

ACM Reference Format:
Li Lin, Jintai Hong, Yanlin Zhuang, and Rongxin Wu. 2026. SmartFuzz: Leveraging Large Language Models and Feature Composition to Generate High-Quality Seeds for Database Fuzzing. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 344 (October 2026), 29 pages. https://doi.org/10.1145/3839476

1 Introduction

Database Management Systems (DBMSs) are fundamental components in data-driven software ecosystems, responsible for efficiently storing, retrieving, and managing massive volumes of structured data [22, 56, 64]. However, due to their immense codebase, intricate query processing logic, and continuous feature evolution, DBMSs are prone to various bugs. Given the central role of

Authors' Contact Information: Li Lin, School of Informatics, Xiamen University, XiaMen, China, linli1210@stu.xmu.edu.cn; Jintai Hong, School of Informatics, Xiamen University, XiaMen, China, hongjintai@stu.xmu.edu.cn; Yanlin Zhuang, School of Informatics, Xiamen University, XiaMen, China, 1159925551@qq.com; Rongxin Wu, School of Informatics, Xiamen University, XiaMen, China, wurongxin@xmu.edu.cn.



ACM 2475-1421/2026/10-ART344
https://doi.org/10.1145/3839476

DBMSs in modern software systems, it is essential to proactively detect and fix these bugs to improve DBMS robustness and data integrity.

Fuzzing is a promising technique for bug detection [8, 26, 29, 33, 45, 53], and it is applied to testing DBMSs [34, 40, 41, 61, 70, 78] by generating SQL (Structured Query Language) queries that contain a series of SQL statements. According to the way SQL queries are produced, DBMS fuzzing approaches can be generally categorized into *generation-based* and *mutation-based* approaches [27, 78]. Generation-based fuzzers [58–61] directly synthesize SQL queries from predefined grammars or abstract syntax trees (ASTs). A representative example is SQLSmith [61], which generates syntactically valid SQL queries by randomly constructing AST. However, generation-based approaches typically lack feedback from program execution and therefore explore the input space in a largely unguided manner. As many SQL statements are processed by shared execution paths inside the DBMS engine, such blind exploration often leads to redundant executions and inefficient state-space coverage [78]. In contrast, mutation-based fuzzing [24, 25, 37, 70, 78] has emerged as one of the most effective techniques for uncovering DBMS bugs. Mutation-based fuzzers maintain a pool of initial seed queries and iteratively generate new test cases by mutating these seeds. The mutated queries are executed on the target DBMS, and feedback such as code coverage [78] or crash signals [72] is used to guide further mutations. By continuously refining test inputs based on runtime feedback, mutation-based fuzzing can explore deeper execution paths and expose subtle bugs.

The effectiveness of mutation-based fuzzing in DBMSs largely depends on the quality of the initial seeds. As the starting point for mutation, initial seeds determine how comprehensively the fuzzer can explore the program's state space. High-quality seeds should not only cover a wide range of DBMS functionalities and SQL constructs, but also contain semantically non-trivial query structures that allow mutations to drive execution into diverse and bug-prone program states [9, 24, 31, 65]. A richer seed corpus naturally leads to higher diversity among mutated inputs, increasing the likelihood of discovering bugs; conversely, if certain syntactic features or semantic behaviors are absent from the initial seeds, bugs associated with those features may remain entirely unexplored. Despite their critical role, most existing mutation-based DBMS fuzzing approaches primarily focus on improving mutation strategies and path exploration techniques, or ensuring that the generated inputs adhere to syntactically and semantically valid SQL formats for deeper logic testing [27]. In contrast, relatively little attention has been paid to the problem of seed selection, which is crucial for enhancing the overall efficiency and depth of DBMS fuzzing.

Existing Efforts. In practice, many DBMS fuzzers rely on manually collected initial seeds extracted from built-in unit tests and regression suites [78, 79]. These seeds are manually curated by DBMS developers and are thus typically syntactically correct and semantically valid, making them reliable starting points for mutation-based fuzzing. However, relying on such test suites as the primary seed source has two key limitations. First, collecting, maintaining, and updating representative SQL cases is time-consuming and error-prone, and many DBMSs do not provide comprehensive or publicly accessible test suites, making it difficult to obtain sufficient seeds for newly developed or less-studied systems. Second, because these seeds are mainly designed for functional correctness verification, they often under-represent rare or complex query patterns and SQL features, leaving a large portion of the feature space under-explored [79].

In this paper, we explore the potential of Large Language Models (LLMs) to guide the generation of initial seeds for DBMS fuzzing. The rapid progress of LLMs such as the Generative Pre-trained Transformer (GPT) family [23] creates a new, lightweight pathway for seed preparation: LLMs can autonomously synthesize executable and semantically valid SQL statements that serve as fuzzing seeds. However, as observed in recent studies such as Fuzz4All [75], LLMs tend to generate SQL statements that closely follow common patterns seen in their training data, leading to limited structural and semantic diversity. Such homogeneity constrains the exploration of rare query

behaviors and deep execution paths that are critical for uncovering complex DBMS bugs. To steer LLMs toward producing high-quality and effective seeds, we propose two strategies:

S1: Document-feature-driven Synthesis. The key idea behind document-feature-driven synthesis is that official SQL documentations encode rich, DBMS-specific syntactic and usage patterns that are not fully captured by LLM pre-training data [41]. Such documentation provides authoritative guidance on how supported SQL features are intended to be used in practice. By leveraging these documentation-derived features as guidance signals for seed synthesis, we generate initial SQL seeds that are syntactically and semantically valid for the target DBMS while covering a broader spectrum of DBMS-specific syntax and functionality. This strategy enables fuzzing to move beyond the distributional biases of pre-trained models and manually curated seeds, thereby improving feature coverage from the very beginning of the fuzzing.

S2: Crash-reuse Synthesis. Our second strategy is motivated by the empirical observation that many DBMS bugs tend to recur across versions, forks, or closely related systems due to extensive code reuse and shared architectural components [41]. As a result, inputs that have previously triggered crashes often encapsulate bug-relevant traits that remain effective beyond a single DBMS version or implementation. Rather than treating historical crashes as isolated incidents, this strategy views them as reusable sources of bug-inducing traits. By recombining such crash-derived characteristics, we aim to synthesize new initial seeds that are more likely to expose latent faults, including previously unseen crash behaviors, while significantly reducing the cost of rediscovering similar bugs from scratch.

Our Approach. Guided by the intuitions behind S1 and S2, we implement SmartFuzz, a practical framework that leverages LLMs to synthesize target-specific, high-quality initial seeds and integrates them into a mutation-based fuzzing pipeline. The workflow of SmartFuzz consists of three sequential stages: feature extraction, LLM-driven synthesis, and compatibility-aware mutation integration. ① **Feature Extraction:** In the feature extraction stage we derive two complementary sets of artifacts. For document features, we collect official DBMS documentation and convert these texts into structured feature records using a lightweight rule-based parsing pipeline. For crash features, we mine historical crash-triggering inputs from bug reports and test logs, minimize each trigger via generalized delta debugging, and compare it with nearby non-crashing variants generated through controlled mutation to extract compact crash-relevant SQL features. ② **LLM-driven Synthesis:** In the LLM-driven synthesis stage, these structured features are assembled into prompts using several ICL-based prompting strategies. The LLM performs feature composition and generalization, synthesizing a set of candidate seeds that are both syntactically correct for the target DBMS and enriched with combinations of crash-relevant traits that increase the likelihood of uncovering latent bugs. ③ **Compatibility-aware Mutation:** Following the design adopted in Sedar [24], SmartFuzz incorporates a compatibility-aware mutation module. This module addresses a practical challenge: SQL seeds synthesized for a target DBMS may contain dialect-specific constructs that cannot be parsed or mutated by off-the-shelf fuzzers, whose mutation engines are typically designed for a fixed and limited SQL grammar. To mitigate this issue, SmartFuzz applies a lightweight compatibility transformation before mutation. Specifically, SQL fragments that are unsupported by the target fuzzer's mutators are conservatively identified and marked as comments. The fuzzer then performs mutations only on the remaining, compatible parts of the query. After mutation, the marked fragments are restored, yielding executable SQL test cases that preserve DBMS-specific semantics. Due to this design, SmartFuzz can be integrated as a plug-and-play seed generator for off-the-shelf DBMS fuzzers, requiring no changes to their mutation engines.

Evaluation. We evaluate SmartFuzz on four widely used open-source DBMSs including MonetDB, Virtuoso, DuckDB, and ClickHouse to assess its effectiveness in real-world mutation-based DBMS

fuzzing. Across 24-hour fuzzing, SmartFuzz detects 61 previously unknown bugs, including 60 bugs that have been confirmed and fixed. Compared with default seeds, local test suites, and cross-DBMS seed transfer, SmartFuzz achieves up to 12.5× higher branch coverage and uncovers 46 additional bugs missed by prior approaches. In summary, this paper makes the following contributions:

- **LLM-guided Seed Synthesis Strategies.** We propose two complementary seed synthesis strategies: document-feature-driven synthesis for covering DBMS-specific functionality, and crash-reuse synthesis for reusing bug-relevant characteristics from historical failures.
- **Practical Framework Design.** We design and implement SmartFuzz, a practical framework that synthesizes feature-rich SQL seeds with LLMs and integrates them into existing mutation-based fuzzers through compatibility-aware mutation.
- **Extensive Evaluation.** We evaluate SmartFuzz on four real-world DBMSs, showing that it improves branch coverage and discovers 61 previously unknown bugs.

2 Background and Motivation

2.1 Preliminaries

DBMS and SQL. A Database Management System (DBMS) is software that stores and retrieves data according to a specified database model. Structured Query Language (SQL) is a domain-specific language used to manage data within a DBMS, and SQL grammar describes both the syntax and semantics of the language. Typically, most DBMS products comply with the basic features of the ANSI SQL standard [1] for common operations but extend support for advanced functionalities through their own unique SQL dialects. However, the grammar of these SQL dialects is not compatible with one another, as similar features might be implemented with distinct grammar variations across different DBMSs. For example, while the character set comparison and sorting function is implemented as "`COLLATION`" in MySQL [52], it is implemented as "`COLLATE FOR`" in PostgreSQL [55]. Such syntactic and semantic heterogeneity limits the portability and scalability of mutation-based fuzzers, since these fuzzers rely on inputs that conform to the specific grammar expected by the target DBMS parser. Moreover, to ensure functional correctness, backward compatibility, and performance stability amid evolving dialect features and implementations, mature DBMSs usually maintain comprehensive test suites, including unit tests and regression tests.

Mutation-based DBMS Fuzzers. Mutation-based fuzzing has been widely recognized as an effective technique for uncovering bugs in software systems. A typical fuzzing system consists of three key components [27]: (1) a seed pool, which serves as the initial set of inputs and evolves throughout the testing process; (2) a mutation engine, which generates new test cases by applying minor modifications or corruptions to the existing seeds, thereby producing diverse inputs to explore different execution paths of the program under test; and (3) an oracle function, which evaluates the outputs or behaviors resulting from the mutated inputs to determine whether they reveal anomalies or security issues and provides feedback for further mutations. However, when fuzzing is applied to DBMSs, it faces unique challenges. DBMSs primarily accept SQL statements as inputs, and the complexity of SQL grammar combined with the intricate logic of DBMS executables makes traditional mutation-based fuzzers ineffective. Inputs generated by these general-purpose, byte-level fuzzers (e.g. AFL [29]) often lead to syntax or semantic errors, preventing deep exploration of the DBMS's internal logic and hindering the discovery of potential bugs.

To overcome these limitations, Squirrel [78] leverages an Intermediate Representation (IR) to structurally maintain the skeleton of SQL statements. Before mutation, the system strips away concrete data, keeping only the operational skeleton. Then, it performs three types of random mutations: (1) inserting type-matched operands; (2) deleting optional operands; and (3) replacing operands with other type-matched ones. This type-based mutation ensures that the generated SQL statements remain syntactically correct, enabling the fuzzer to reach the deep logic of the DBMS.

Table 1. **Summary of Seed Sources.**

Paper	Venue	Seed
SQUIRREL [78]	Security'20	unit tests from official GitHub repository
MUTASQL [10]	DBTest'20	manually constructed
SQLRIGHT [39]	Security'22	unit tests from official GitHub repository
EQSQL [72]	Appl.Sci.'23	unit tests from official GitHub repository
GRIFFIN [25]	ASE'22	official regression test suites
LEGO [37]	ICDE'23	unit tests from official GitHub repository
SEDAR [24]	ICSE'24	transferring test cases from other DBMSs
SQLASER [71]	ArXiv'24	unit tests from official GitHub repository

2.2 Seed Selection Practices

Importance of Initial Seeds. In DBMS fuzzing, a seed typically refers to a test case composed of a sequence of SQL statements. Mutation-based DBMS fuzzing continuously mutates existing seeds to generate new test cases. The quality of these initial seeds is crucial to the overall effectiveness of fuzzing. First, initial seeds serve as the foundation and starting point of the fuzzing process, upon which subsequent iterations are built. The richness and diversity of features embedded in the initial seeds directly affect the range of DBMS functionalities that can be explored through later mutations [24]. High-quality seeds can thus improve code coverage and significantly enhance the likelihood of uncovering hidden bugs and logic flaws within the DBMS.

Empirical Study of Seed Selection Practices. As shown in Table 1, we further examine eight mutation-based DBMS fuzzing papers published after 2020 (following SQUIRREL [78]) to analyze the sources of their seed inputs. The table reveals a clear trend: the majority of existing approaches extract initial seeds from official DBMS test artifacts, including unit tests hosted in public GitHub repositories and built-in regression test suites. These SQL test cases are manually curated by DBMS developers and are therefore syntactically correct and semantically valid, making them suitable starting points for mutation-based fuzzing.

Limitation of Existing Seed Selection Strategies. While DBMS-maintained test suites provide reliable and semantically valid SQL seeds, relying on them as the primary seed source introduces some limitations. Manually collecting, maintaining, or writing such test cases is time-consuming and error-prone, and many DBMSs do not maintain comprehensive or publicly accessible test suites. This makes it difficult to obtain sufficient high-quality seeds for fuzzing newly developed, rapidly evolving, or less-studied systems. To alleviate this dependency, SEDAR [24] proposes transferring test cases across different DBMSs by reusing manually written tests from well-tested systems. However, these test suites are primarily designed for functional correctness verification resulting in limited coverage of rare or complex query patterns and SQL features [79]. Moreover, due to SQL dialect differences, transferred tests may not cover the target DBMS's diverse, system-specific features. Motivated by these limitations, our work aims to automatically synthesize high-quality, diverse, and crash-informed seeds that can more effectively explore complex DBMS execution paths and uncover hidden bugs.

2.3 Design Objective and Key Insight

Design Objectives. The goal of this work is to design an automated seed synthesis system that improves the effectiveness of mutation-based DBMS fuzzing. Rather than building a new DBMS fuzzer, we focus on systematically constructing high-quality initial SQL seeds that can be seamlessly integrated into existing fuzzing pipelines.

Specifically, we aim to automatically construct high-quality initial SQL seeds that (i) are syntactically and semantically valid for the target DBMS, (ii) incorporate rich and diverse SQL features that facilitate exploration of a wide range of DBMS execution paths, and (iii) encode bug-relevant

characteristics that increase the likelihood of triggering security-critical failures. Moreover, the synthesized seeds should be reusable by existing mutation-based fuzzers without requiring modifications to their mutation engines, ensuring compatibility, scalability, and practical applicability in real-world fuzzing environments.

Key Insights. LLMs offer a mechanism for synthesizing valid SQL queries at scale, making them a promising candidate for automated seed construction. However, naive LLM generation tends to reproduce frequent SQL patterns seen during pre-training, yielding low structural diversity and weak coverage of rare features that are critical for fuzzing DBMSs. To unlock the potential of LLM-based seed synthesis, our design is guided by the following two key insights.

Insight 1: Documentation Defines the DBMS Feature Space. First, official DBMS documentation constitutes an underutilized source of high-quality, DBMS-specific knowledge. Such documentation explicitly describes supported SQL features, usage constraints, and illustrative examples, which together define the practical feature space of a DBMS but are often underrepresented in LLM training data. By extracting and leveraging these documented features, we can guide seed synthesis toward feature-rich and diverse SQL inputs beyond common or repetitive query patterns.

Insight 2: Historical Crashes Encode Bug-Relevant Traits. Second, historical crash-triggering inputs encode concise and actionable bug-relevant characteristics. Due to code reuse and shared implementation logic, many DBMS bugs recur across versions or closely related systems. By mining and recombining these crash-derived features, we can synthesize seeds that inherit bug-triggering potential and are more likely to expose latent vulnerabilities.

Together, these insights motivate a synthesis strategy that combines documented feature coverage with crash-informed guidance, enabling the automated construction of effective initial seeds for DBMS fuzzing.

2.4 Motivating Example

Document-Feature-Driven Seed Synthesis Enables Previously Unreachable Crash Paths. Figure 1 illustrates a crash bug in MonetDB (GitHub Issue #7709 [2]) that remains undetected by existing mutation-based DBMS fuzzers due to the absence of appropriate initial seeds. This bug has been assigned CVE-2025-69761. Specifically, the query "`FIELD('bar', (SELECT name FROM test_fields WHERE id = 2))`" is syntactically valid and semantically well-defined, yet causes MonetDB to abort with an internal assertion failure during expression binding. This behavior indicates that the DBMS reaches an unhandled execution state when expanding a subquery expression as an argument to the "`FIELD`" function, revealing a flaw in its internal expression processing logic rather than an explicitly unsupported SQL construct. However, this bug is difficult for existing DBMS fuzzers to detect, not because of insufficient mutation capability, but because the required feature—the "`FIELD`" function and its associated usage patterns—is absent from their initial seed corpora, including official unit tests and regression suites. When a feature does not appear in the seed pool, mutation-based fuzzers cannot synthesize it through random insertion or replacement, rendering the corresponding execution paths unreachable regardless of fuzzing duration. In contrast, our document-feature-driven synthesis explicitly extracts the "`FIELD`" function from SQL documentation and injects it into the initial seed pool, enabling subsequent mutations to explore combinations such as "`FIELD`" with subquery arguments and thereby making this previously unreachable crash path detectable.

Crash-Reuse Synthesis Discovers New Bugs via Feature Recombination. Figure 2 illustrates a memory-safety crash in DuckDB (Github Issue #20615 [16]) triggered by a synthesized seed produced via crash-reuse synthesis. The crash is caused by a subtle type/behavior mismatch in the `LAG` window function: when `LAG` is applied to a `VARCHAR` column, an out-of-range offset forces the execution to return the default value; however, if the default value's type does not match

```
CREATE TABLE test_fields (id INT, name VARCHAR(50));
INSERT INTO test_fields VALUES (1, 'foo'), (2, 'bar'), (3, 'baz'), (4, 'qux');

SELECT field('bar', (SELECT name FROM test_fields WHERE id = 2)) AS
index_in_subquery;
rel_bin.c:<line>: Assertion `s' failed.   In MonetDB: server crashed
```

Fig. 1. **Motivating Example 1: A crash bug in MonetDB found by SmartFuzz.** Document-guided seed synthesis makes the `FIELD`-subquery crash reachable

```
pragma enable_verification;
CREATE TABLE v0 ( v1 INTEGER , v3 VARCHAR(16) , v2 INTEGER ) ;
INSERT INTO v0 VALUES ( 0 , 1 , 1 ) ;
INSERT INTO v0 VALUES ( 1 , NULL , 2 ) ;

SELECT LAG ( v3 , 10 , 3 ) OVER( ) FROM v0 ;
AddressSanitizer: wild pointer access.   In DuckDB: process crashed
```

Fig. 2. **Motivating Example 2: A crash bug in DuckDB found by SmartFuzz.** Crash-reuse synthesis recombines a historical `LAG` bug feature (from SQLite) to trigger a new DuckDB crash.

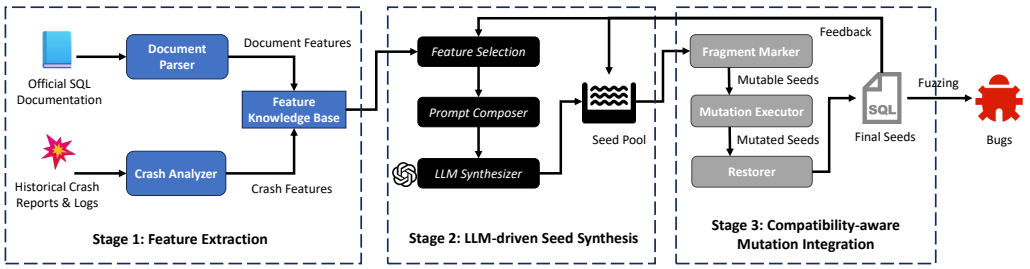


Fig. 3. **The overall workflow of SmartFuzz**

the input column type, the executor may incorrectly treat the default as a string value, leading to a wild pointer access during internal string-copying logic. This bug is found by reusing and recombining historical crash features. Specifically, SmartFuzz starts from a historical SQLite bug report (Ticket `d87336c81c` [3]) and extracts the `LAG` crash feature. It then prompts an LLM to compose this crash feature with additional window-related features, `OVER()` window specification, to generate new executable seeds that systematically explore the interaction space between `LAG` and window execution. By injecting these crash-reused, recomposed seeds into the fuzzing seed pool, subsequent mutations quickly exercise the vulnerable execution path in DuckDB and expose the crash. This example demonstrates the advantage of crash-reuse synthesis: historical bug features can be treated as reusable building blocks, and their recombination with other features can surface new vulnerabilities that are difficult to reach through unguided mutation alone.

3 Approach

3.1 Overview

Figure 3 shows the overall workflow of SmartFuzz, an automated seed synthesis framework designed to enhance mutation-based DBMS fuzzing across DBMSs. SmartFuzz follows a three-stage pipeline that systematically constructs high-quality initial SQL seeds and integrates them into existing

fuzzing workflows. This staged design follows the task-decomposition principle commonly used in modern agentic workflows: instead of relying on a single monolithic prompt, SmartFuzz decomposes the complex goal of DBMS seed synthesis into specialized and manageable stages. Although LLMs can generate valid SQL, they tend to follow common pretraining patterns and may under-exercise rare DBMS-specific features or bug-prone structures [11, 41]. Thus, SmartFuzz organizes the workflow into feature extraction, feature-guided seed synthesis, and compatibility-aware mutation integration, allowing the LLM to focus on feature composition while mutation-based fuzzers perform efficient coverage-guided exploration.

- **Stage 1: Feature Extraction.** SmartFuzz extracts structured features from two complementary sources. Official SQL documentation is processed to derive DBMS-specific SQL features and intended usage patterns, while historical crash reports and logs are analyzed to extract bug-relevant features from past failures. These documentation-derived and crash-derived features are consolidated into a Feature Knowledge Base, which serves as the input to seed synthesis.
- **Stage 2: LLM-driven Seed Synthesis.** Based on the extracted features, SmartFuzz synthesizes initial seeds using an LLM. A feature selection component samples relevant documentation and crash features, assembled into feature-aware prompts by a prompt composer. Guided by these prompts, the LLM synthesizer generates candidate SQL queries syntactically and semantically valid for target DBMSs. Valid queries are collected into a seed pool for subsequent fuzzing.
- **Stage 3: Compatibility-aware Mutation Integration.** SmartFuzz integrates the synthesized seeds into a mutation-based fuzzing pipeline. Since off-the-shelf DBMS fuzzers support only a subset of SQL grammar, dialect-specific fragments are first identified and temporarily commented out to form mutable seeds that can be parsed and mutated. Structural mutations are then applied to the parsable regions, after which the commented fragments are restored to obtain executable final seeds for fuzzing. The final seeds are executed on the target DBMS, and crash and coverage feedback is collected for downstream refinement.

Through this design, SmartFuzz bridges documentation knowledge, crash reuse, and practical fuzzing constraints, enabling effective and scalable seed synthesis for mutation-based fuzzing.

3.2 Feature Extraction

Document Parser. To enrich the feature diversity of initial seeds, SmartFuzz systematically extracts documented SQL features and leverages them to guide seed synthesis toward feature-rich and diverse query structures. Different from QTRAN [41], SmartFuzz aims to capture a broad spectrum of SQL constructs described in official documentation, rather than focusing on a limited subset of frequently used elements. Extracted features include functions, data types, operators, keywords, clauses, and statement-level constructs, as well as DBMS-specific extensions. By incorporating such documented features into the seed space, SmartFuzz exposes mutation-based fuzzers to a wider range of syntactic and semantic behaviors.

For each target DBMS, SmartFuzz employs a lightweight parser to extract feature information directly from official SQL reference pages. The parser identifies feature entries based on the documentation's structural organization (e.g., categorized reference sections, feature indices, or statement descriptions) and retrieves the corresponding HTML pages. From each page, the parser extracts three core components when available: (1) the feature specification or grammar rule, (2) a natural-language description of intended semantics and usage constraints, and (3) example SQL snippets illustrating typical usage. Code-related elements such as grammar rules and examples are extracted from structured HTML tags, while the remaining textual content is treated as semantic explanation. In practice, official documentation may omit explanations or example usages for certain features. Instead of discarding such partially specified features, SmartFuzz leverages an

Algorithm 1: Crash Feature Extraction

Input: F: a crash-inducing SQL statement
Output: C: a set of extracted crash features

$C \leftarrow \emptyset$; // Extracted crash features
$\mathcal{P} \leftarrow \emptyset$; // Passing variants
Step 1: Crash-Preserving Reduction;
$F_{min} \leftarrow$ Reduce(F)
Step 2: Passing Variant Generation;
$\mathcal{M} \leftarrow$ Initial LocalPerturbations ; // A set of mutation rules
foreach $m \in \mathcal{M}$ **do**
  if *Applicable*(m, F_{min}) **then**
    $F' \leftarrow$ ApplyMutation(F_{min}, m);
    if *IsValid*(F') ***and*** *DoesNotCrash*(F') **then**
      $\mathcal{P} \leftarrow \mathcal{P} \cup \{F'\}$;
Step 3: Crash Feature Abstraction;
foreach $F' \in \mathcal{P}$ **do**
  $\Delta \leftarrow$ SyntacticSemanticDiff(F_{min}, F');
  $C \leftarrow C \cup$ AbstractFeatures(Δ);
return C;

LLM to automatically synthesize missing explanations or examples based on the available feature specification and surrounding context. This completion step allows SmartFuzz to preserve rare and sparsely documented features in the knowledge base, thereby reducing bias toward well-documented constructs and further enriching feature diversity. All extracted and completed features are normalized and stored in a unified Feature Knowledge Base, which serves as a key input for subsequent LLM-driven seed synthesis.

Crash Analyzer. Motivated by the empirical observation that many DBMS bugs recur across versions, forks, or closely related systems due to extensive code reuse and shared architectural components [41], historical crash-inducing SQL statements provide a valuable source of bug-relevant knowledge. However, directly reusing such inputs as fuzzing seeds is often ineffective, as they are typically highly coupled to particular configurations, limiting their ability to generalize to nearby failure-inducing states. To overcome this limitation, SmartFuzz extracts fine-grained crash features from historical crash-triggering SQL statements, rather than treating each crashing input as an atomic seed. These features capture reusable bug-relevant characteristics that can be recombined during LLM-guided seed synthesis, enabling systematic exploration of latent bugs beyond simple crash replay. Algorithm 1 describes how SmartFuzz extracts fine-grained crash features from historical crash-inducing SQL statements.

Step 1: Crash-Preserving Reduction. Given a crash-inducing SQL program F, the algorithm first applies SQL-aware delta debugging to obtain a minimized crashing input F_{min}. Delta debugging [7, 40, 47, 57] significantly reduces the syntactic complexity of the input while preserving the crash behavior, thereby narrowing the search space for subsequent analysis. In our implementation, we adopt existing SQL reduction techniques SQLess [40] to perform this step. Although F_{min} still

triggers the crash, it may contain auxiliary syntactic or semantic elements that are required only for query validity rather than being causally related to the bug.

Step 2: Passing Variant Generation. To isolate bug-relevant characteristics, the algorithm then constructs a set of passing variants by applying a curated set of mutation rules $\mathcal{M}$ to F_{min}. Each mutation rule $m \in \mathcal{M}$ introduces a small, localized modification to the query while preserving overall syntactic validity. We design four categories of mutation rules to minimally perturb different components of SQL queries:

- *Identifier-level Mutations.* These mutations modify schema-related identifiers and collation settings, such as replacing table or column references with constant expressions, removing or substituting `COLLATE` clauses, or switching character sets.
- *Operator-level Mutations.* These mutations modify how expressions are evaluated by substituting arithmetic, comparison, or type-related operators. For example, replacing division with integer division, switching between equality-based and null-aware comparison operators, or adjusting operator precedence.
- *Structure-level Mutations.* These mutations simplify or reorganize the syntactic structure of a query while preserving its high-level intent. Examples include removing redundant parentheses, flattening nested `SELECT` subqueries into a single-level query, or replacing `UNION` with `UNION ALL`.
- *Clause- and Statement-level Mutations.* These mutations remove, or substitute higher-level SQL constructs, such as `HAVING` clauses, window functions, `LATERAL` joins, `LIMIT/OFFSET`, or different join variants.

These mutation rules are provided to the LLM as few-shot exemplars, guiding it to synthesize passing variants that apply such localized perturbations while preserving SQL validity and executability. If a mutated program F' remains executable but no longer triggers the crash, it is retained as a passing variant. These passing variants serve as contrastive examples that differ minimally from the crashing input while avoiding the failure.

Step 3: Crash Feature Abstraction. Finally, the algorithm compares F_{min} with each passing variant at the AST level to identify crash-relevant syntactic and semantic differences. Specifically, it aligns structurally corresponding AST nodes and extracts node-level changes, such as inserted, deleted, or replaced operators, clauses, expressions, and execution-related settings. Because each passing variant differs from F_{min} through a controlled mutation, the changed nodes that coincide with the transition from crashing to passing are retained as candidate crash features, together with their surrounding syntactic contexts. The algorithm further normalizes query-specific identifiers and literal values while preserving their semantic categories and structural relationships, making the extracted features reusable across different SQL programs. These features guide subsequent LLM-based synthesis to recombine bug-relevant constructs in diverse contexts and systematically explore execution states related to historical crashes.

3.3 LLM-driven Seed Synthesis

Feature Selection. Given a feature knowledge base that contains both documentation-derived features and crash-derived features, a key design question is how to select and combine features when constructing prompts for LLM-driven seed synthesis. Selecting which features is critical to achieving a balance between diversity and validity when synthesizing SQL seeds. A simplistic approach that feeds a single feature into the LLM at a time may produce focused queries, but it risks missing interesting feature interactions that only manifest when multiple constructs are combined. Conversely, blindly combining many features can lead to overly complex and semantically incorrect statements. To navigate this trade–off, SmartFuzz uses two concrete scheduling strategies:

- **Controlled combination.** SmartFuzz regulates the number of features in each prompt through a three-stage combination policy. It begins with single-feature prompts to establish seed validity during an initial "cold start" phase. Once a sufficient number of valid seeds have been synthesized, the sampler transitions to two-feature prompts, which form the main exploration phase and expose feature interactions. Three-feature prompts are occasionally attempted to probe higher-order interactions but are rate-limited to avoid overwhelming the LLM and degrading synthesis validity. In our implementation, these three stages account for 30%, 50%, and 20% of the synthesis budget, respectively. This staged policy constrains prompt complexity while allowing the system to gradually explore richer feature interactions.
- **Feature coverage-driven adaptation.** SmartFuzz maintains lightweight coverage statistics during synthesis, tracking (i) how often each feature has been used in prompt construction and (ii) whether a feature has appeared in a successfully executed or mutated SQL seed. Features that have not yet been exercised, or that appear infrequently, are prioritized for subsequent sampling, whereas feature pairs that have been repeatedly used are temporarily deprioritized to reduce redundancy. The scheduler periodically updates these statistics, allowing the system to shift attention toward underrepresented constructs and avoid stagnation. Concretely, after every 100 synthesized seeds, we compute for each feature its prompt usage count and valid-seed count, and sort all features in ascending order of the sum of these two counts. In the next sampling round, features from the lowest-ranked 30% are preferentially selected.

The two strategies are complementary: controlled combination determines how many features to include in a prompt, while coverage-driven adaptation determines which features should be selected within that budget. Together, they enable SmartFuzz to explore a large synthesis space without compromising SQL validity or generation throughput.

Prompt Composer. Once a set of features is selected, SmartFuzz assembles them into a structured prompt that specifies the synthesis task to the LLM. The prompt follows an instruction-style ICL format, where the model is given an explicit role, goal, and a set of feature descriptions that serve as conditioning context. Figure 4 illustrates the concrete prompt template used by SmartFuzz. The prompt template consists of four components: System Role, Goal, Knowledge Input, and Synthesis Constraints. The System Role declares the model as a "DBMS fuzzing-oriented SQL synthesis agent and SQL semantics expert", steering it toward fuzzing-oriented synthesis rather than generic text-to-SQL translation. The Goal specifies the task of generating "semantically rich and execution-path-diverse SQL seeds" for a target DBMS. The Knowledge Input block enumerates the selected features, where each feature record includes its type (e.g., operator, function, clause, crash-feature), natural-language semantics, and usage examples. This structured conditioning enables the LLM to actively exercise the provided constructs rather than merely recalling SQL templates from pre-training data. Finally, the Synthesis Constraints describe fuzzing-oriented requirements, including cross-feature composition (all features must appear and preferably interact), semantic stress (e.g., implicit casting, `NULL` propagation, evaluation order), structural complexity (e.g., joins, subqueries, grouping, windowing), validity and diversity, and a machine-parsable output format.

LLM Synthesizer. After constructing the prompts, SmartFuzz enumerates them and queries the underlying LLM to synthesize candidate SQL seeds. Since each prompt is self-contained and does not rely on external conversational context, prompts can be issued independently and evaluated in parallel to improve throughput. The LLM returns a JSON-formatted list of SQL statements, which are then passed through a lightweight filtering stage to validate syntactic correctness and basic executability on the target DBMS. Valid statements are inserted into the seed pool for subsequent mutation-based fuzzing, while malformed or semantically invalid candidates are discarded.

Prompt Composer

System Role:
You are a DBMS fuzzing-oriented SQL synthesis agent and SQL semantics expert.

Goal:
Generate semantically rich and execution-path-diverse SQL seeds for fuzzing {db_name}.

Knowledge Input (Feature-Conditioned):
The following features are provided and must be actively used:
{Feature_1}、{Feature_2} … {Feature_i} {
type: {operator | function | clause | crash-feature | …}
semantics: ...
usage examples: ...
}

Synthesis Constraints:
1、Cross-Feature Composition
- All provided features must appear in the generated SQL.
- Prefer nested and interacting usage (e.g., operators applied to function outputs, ...).
2、Semantic Stress
- Exercise edge execution behaviors, including implicit casting, NULL propagation, precedence, evaluation order, or crash-relevant semantics (...).
3、Structural Complexity
- Incorporate non-trivial query structures such as joins, subqueries, grouping, windowing, and expression reuse across clauses (...).
4、Validity & Diversity
- SQL must be syntactically valid and semantically executable on {db_name}.
- Seeds should vary in both structural and semantic dimensions.
5、Output Format:
- Output strictly as a JSON array and each element is a complete SQL statement ending with ';'.

Fig. 4. **Prompt template for feature-guided SQL synthesis**

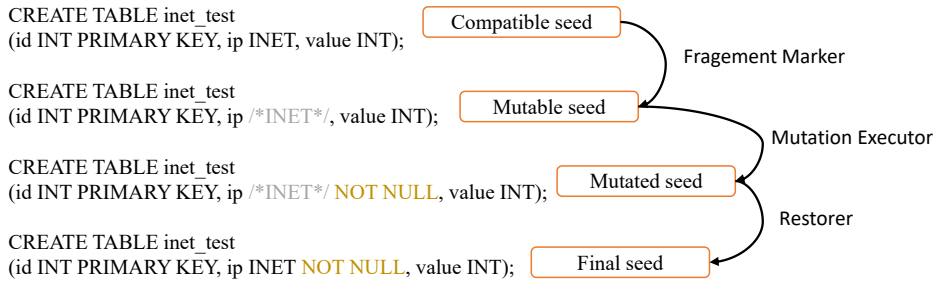


Fig. 5. **An example of compatibility-aware mutation**

3.4 Compatibility-aware Mutation Integration

Although synthesized seeds are executable on the target DBMS, they may contain dialect-specific constructs or extensions that cannot be parsed or mutated by existing fuzzers, whose mutators typically support only a subset of SQL grammar. Such incompatibility prevents the mutator from constructing intermediate representations for mutation, thereby limiting fuzzing effectiveness. To address this issue, SmartFuzz incorporates a compatibility-aware mutation module that adapts synthesized seeds for off-the-shelf mutation-based DBMS fuzzers following SEDAR's [24] design. As SEDAR's implementation is not publicly available, we reimplemented this module for integration into our pipeline. As illustrated in Figure 5, the refinement process follows a *mark–mutate–restore* workflow. Given a dialect-specific compatible seed, SmartFuzz first identifies unsupported SQL fragments and temporarily comments them out to form a mutable seed that can be parsed and structurally mutated by the off-the-shelf mutator. This identification is fully rule-based and does not involve an LLM: SmartFuzz runs the parser in error-recovery mode, which records unrecognized token ranges while continuing to parse the remaining input, and marks the corresponding SQL fragments as unsupported. The mutator then applies structural mutations on the parsable regions, producing a mutated seed. Finally, the commented fragments are restored to yield a final seed that preserves dialect-specific semantics while incorporating structural mutations for fuzzing. This refinement enables synthesized seeds to be reused by existing fuzzers without modifying their parsers or grammars.

3.5 Implementation Details

SmartFuzz contains 2,128 lines of Python and 112,522 lines of C++. Among the C++ code, 1,914 lines are new contributions, while the remaining 110,608 lines are inherited from SQUIRREL. As shown in Figure 3, the system consists of three modules: Feature Extraction, LLM-driven Seed Synthesis, and Compatibility-aware Mutation Integration. For feature extraction, we collected documentation-derived SQL features from four target DBMSs (MonetDB [48], Virtuoso [51], DuckDB [20], ClickHouse [12]) and gathered crash-inducing SQL statements from eight DBMS bug repositories (MonetDB [48], ClickHouse [12], DuckDB [20], MySQL [14], OceanBase [15], PostgreSQL [30], Virtuoso [51], and SQLite [13]). For LLM-driven seed synthesis, we employ GPT-4o-mini [49] as the underlying model. The LLM composes selected features into prompts and synthesizes candidate SQL queries via an online API. Synthesized queries are checked for syntax and basic executability, and valid ones form the initial seed pool. For compatibility-aware mutation integration, we integrate the synthesized seeds with SQUIRREL [78], an open-source mutation-based DBMS fuzzer. SQUIRREL has been shown to find numerous bugs across several DBMSs and achieve higher semantic correctness than prior fuzzers. We choose SQUIRREL both for its popularity and to align with SEDAR's experimental setup. GRIFFIN [25], although used in SEDAR, was excluded due to lack of source availability. SQUIRREL's parser is implemented using Flex and Bison. To support dialect-aware mutation, we extend the Bison grammar with fine-grained error recovery: when an unsupported dialect fragment is encountered, the parser records its start and end offsets. The caller then inserts comment delimiters at these positions to produce a mutable query, applies structural mutations using Squirrel's mutation engine, and finally restores the commented fragments to obtain an executable seed for fuzzing.

Effort of Adaption. Adapting SmartFuzz to a new DBMS primarily involves extending the generic parser to support dialect-specific syntax and implementing a client interface for query execution and schema retrieval. Because the compatibility-aware mutation layer decouples dialect handling from the fuzzing engine, no modifications to the core fuzzing module are typically required. In our experiments, one author required roughly eight hours to port SmartFuzz to a new DBMS.

4 Evaluation

We evaluate SmartFuzz to answer the following research questions:

- **RQ1:** Can SmartFuzz help mutation-based fuzzers discover new bugs in real-world DBMSs? This question evaluates the practical effectiveness of our synthesized seeds in terms of bug discovery.
- **RQ2:** How does SmartFuzz compare against existing seed sources? This question examines whether SmartFuzz produces higher-quality initial seeds in DBMS fuzzing.
- **RQ3:** Does extracting fine-grained crash features from historical bugs improve fuzzing effectiveness compared to directly reusing historical crash-inducing SQL inputs? This question evaluates the contribution of the crash analyzer module, testing whether decomposing and recombining crash-relevant features enables deeper exploration than simple crash replay.

Tested DBMSs. We evaluate SmartFuzz on four widely used open-source DBMSs, as summarized in Table 2. These systems are selected based on three criteria: (*i*) popularity and adoption in industry and academia, (*ii*) diverse SQL dialects and execution semantics, and (*iii*) active development communities with sustained maintenance. In addition, all four DBMSs have been extensively evaluated in prior DBMS fuzzing studies such as SEDAR [24].

Environment. We conduct the experiments on one server with 104-cores Intel(R) Xeon(R) Gold 6230R CPU @2.10GHz and 512 GB memory. The server operates under the Ubuntu 20.04 OS, with the 5.4.0-135-generic version of the Linux kernel.

Table 2. **Demographics of the DBMSs Under Test**

DBMS	Version	GitHub Stars	DB-Engines Rank	First Release
MonetDB	v11.54.0	449	153	2004
Virtuoso	v7.2.17	6.1K	91	1998
DuckDB	v1.4.1	35.3K	42	2018
ClickHouse	v23.5.3.4	45.1K	30	2016

Table 3. **Summary of Bugs Discovered by SmartFuzz**

DBMS	Found	Confirmed	Fixed	Waiting
MonetDB	32	32	32	0
Virtuoso	20	19	19	1
DuckDB	8	8	8	0
ClickHouse	1	1	1	0
Total	61	60	60	1

4.1 Detected Bugs

4.1.1 Setup. We use SmartFuzz to fuzz each of the four DBMSs for 24 hours independently. During fuzzing, all observed crashes and abnormal errors are collected, deduplicated, and post-processed before being reported to the developers for validation. For each crash-triggering query, we apply SQLess [40] to perform crash-preserving minimization. In some cases, the developers assisted us in further minimizing the triggering programs using their system-specific knowledge.

4.1.2 Results. **New Bugs.** Table 3 summarizes the bugs discovered by SmartFuzz across the four evaluated DBMSs under the same 24-hour fuzzing budget per system. Overall, SmartFuzz uncovers 61 previously unknown bugs, including 32 in MonetDB, 20 in Virtuoso, 8 in DuckDB, and 1 in ClickHouse. At the time of submission, 60 of these bugs have been confirmed and fixed by developers.

Bug Type Diversity. Table 4 shows that SmartFuzz uncovers diverse failure types across the four DBMSs, indicating that our synthesized seeds can drive executions into a wide range of bug-prone states rather than overfitting to a single crash pattern. Overall, the 61 bugs span 7 categories, including assertion failures, null pointer dereferences, heap/stack buffer overflows, segmentation violations, abnormal errors, and double free. Assertion failures dominate the results (31/61), especially in MonetDB (27/32), suggesting that many generated workloads exercise corner-case execution paths that violate implicit internal invariants. Importantly, SmartFuzz also reveals multiple memory-safety-related bugs—including 6 buffer overflows, 3 null pointer dereferences, 9 segmentation violations, and a double-free bug—which are typically security-critical and may be exploitable. In addition, abnormal errors appear frequently and across multiple systems, reflecting SmartFuzz's ability to trigger complex, DBMS-specific behaviors beyond simple assertion failures.

Statement Distribution of Bug-triggering Queries. Figure 6 presents the distribution of SQL statement types appearing in queries that trigger DBMS bugs. `CREATE TABLE` is the most prevalent statement, appearing in nearly all bug-triggering queries, as it establishes the base relations required by subsequent operations. `INSERT` and `SELECT` are also highly frequent, indicating that data population and query execution play a central role in driving the DBMS into bug-prone states. In particular, `SELECT` statements often serve as the final statement in a query sequence, where accumulated state interactions are ultimately exercised and exposed. Other statements such as `UPDATE`, `ALTER`, `DROP`, and `CREATE VIEW` appear less frequently, but are important for mutating

Table 4. **Distribution of Discovered Bugs by DBMS and Bug Type.**

DBMS	AF	NPD	HBOF	SBOF	SEGV	AB	DF	Total
MonetDB	27	1	1	1	0	2	0	32
Virtuoso	3	1	3	1	9	2	1	20
DuckDB	1	0	2	0	0	5	0	8
ClickHouse	0	1	0	0	0	0	0	1
Total	31	3	6	2	9	9	1	61

Notes. AF: Assertion Failure; NPD: Null Pointer Dereference; HBOF: Heap Buffer Overflow; SBOF: Stack Buffer Overflow; SEGV: Segmentation Violation; AB: Abnormal Error; DF: Double Free.

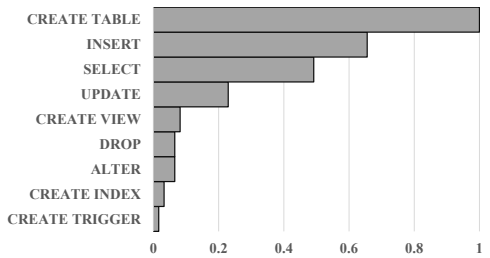


Fig. 6. **Percentage of queries including specific statements that trigger DBMS bugs.**

schema or data states and enabling complex execution scenarios. Although statements like `CREATE INDEX` and `CREATE TRIGGER` occur rarely, their presence shows that bugs can also be triggered by advanced or less commonly tested DBMS features. Overall, the distribution suggests that DBMS bugs are typically exposed through multi-statement queries that combine schema construction, data manipulation, and query execution, rather than isolated single-statement workloads.

Bug Feature Diversity. To further understand the effectiveness of our synthesized seeds, we analyze the feature diversity of the bugs uncovered by SmartFuzz. As shown in Table 5, rather than concentrating on a small set of recurring SQL patterns, the discovered bugs span a wide range of bug-triggering features across different DBMSs. These features include not only common constructs such as subqueries, aggregation, and joins, but also many less frequently tested yet error-prone features, such as window functions, recursive CTEs, CHECK constraints, NATURAL/FULL OUTER JOINs, ALTER TABLE variants, and advanced analytic functions. We further classify these bug-triggering features into common SQL features and DBMS-specific/rare features, and mark whether they are covered by existing seed corpora. Among the 61 discovered bugs, 45 are triggered by DBMS-specific or rare features, while 16 involve common SQL features. Moreover, 27 bug-triggering feature sets are not covered by existing seed corpora. Moreover, several bugs are triggered by complex combinations of features instead of a single isolated construct, suggesting that our seeds can effectively drive the fuzzer toward semantically rich and non-trivial execution paths. These results indicate that SmartFuzz does not merely improve bug finding by generating more seeds, but by expanding the feature space exposed to mutation-based fuzzers. Such diversity is important for mutation-based fuzzing, because it increases the chance of exploring under-tested DBMS components and exposing deeply hidden implementation faults.

Feature Coverage of Existing Seed Corpora. To quantify the gap between existing seed corpora and SmartFuzz-generated seeds, we also conduct a feature-level analysis over documented DBMS features. We compare documented DBMS features covered by Squirrel's default seeds, official unit/regression tests, and SmartFuzz seeds. SmartFuzz introduces 360, 723, 519, and 1,231 additional documented features over Squirrel's default seeds on MonetDB, Virtuoso, DuckDB, and

Table 5. **Diversity of Bug-Triggering Features**

Id	Bug Description	Bug Status	Key Features	Class	Seed Cov.
MonetDB					
1	Crash in field() with subquery argument	fixed (#7709)	field, subquery	Dialect	No
2	RPAD crashes on huge CHAR length	fixed (#7710)	rpad, CHAR datatype	Dialect	No
3	GROUP BY outside projection crashes optimizer	fixed (#7713)	GROUP BY, count	Standard	Yes
4	Window function crashes CHECK constraint handling	fixed (#7714)	DENSE_RANK, CHECK	Dialect	Yes
5	CHECK with window function triggers exp_bin crash	fixed (#7717)	STDDEV_SAMP, CHECK	Dialect	Yes
6	Constant window bounds crash LAST_VALUE handling	fixed (#7719)	LAST_VALUE, constant bounds	Standard	No
7	UPDATE subquery missing single-row validation crashes	fixed (#7735)	JSON_OBJECTAGG, PERCENT_RANK, NTH_VALUE	Dialect	No
8	Negative scale causes integer overflow	fixed (#7736)	reduce_scale, JSON_ARRAYAGG	Dialect	No
9	DISTINCT join window query breaks rel_name	fixed (#7741)	JSON_OBJECTAGG, DISTINCT, BIT_OR	Dialect	No
10	Complex DISTINCT query dereferences NULL	fixed (#7742)	recursive CTE, PERCENT_RANK, DENSE_RANK	Dialect	No
11	Scalar grouped subquery crashes aggregate compilation	fixed (#7766)	subquery, GROUP BY	Standard	Yes
12	Malformed IN subquery breaks rel2bin_select	fixed (#7767)	IN subquery	Dialect	Yes
13	NTH_VALUE crashes under RANGE frame	fixed (#7769)	NTH_VALUE, RANGE frame	Standard	Yes
14	Aggregated scalar subquery crashes INSERT VALUES	fixed (#7770)	subquery, HAVING	Dialect	Yes
15	Scalar subquery breaks INSERT aliasing	fixed (#7771)	subquery, INSERT VALUES	Dialect	Yes
16	Renaming checked table triggers assertion failure	fixed (#7774)	CHECK, ALTER TABLE RENAME	Standard	Yes
17	LEAD query triggers heap buffer overflow	fixed (#7775)	LEAD, subquery	Dialect	Yes
18	Recursive UNION ALL enters infinite loop	fixed (#7785)	recursive CTE, UNION ALL	Standard	Yes
19	Dropping renamed table exhausts memory	fixed (#7786)	ALTER TABLE RENAME, DROP TABLE	Standard	Yes
20	Invalid CHECK crashes ALTER TABLE ADD COLUMN	fixed (#7787)	CUME_DIST, CHECK, ALTER TABLE ADD COLUMN	Dialect	Yes
21	Windowed UPDATE subquery breaks rel_order_by	fixed (#7788)	NTH_VALUE, subquery, UPDATE	Dialect	Yes
22	Subquery inside CHECK crashes exp_bin	fixed (#7789)	CHECK, subquery	Dialect	Yes
23	Windowed CTAS query breaks rel2bin_insert	fixed (#7790)	LAG, CTAS	Standard	No
24	Aggregate comparison breaks rel2bin_groupjoin	fixed (#7791)	CASE expression, subquery	Dialect	Yes
25	Nested NOT IN subquery breaks join lowering	fixed (#7792)	NOT IN, nested subquery	Dialect	Yes
26	UPDATE IN-subquery breaks semijoin lowering	fixed (#7793)	IN subquery, DISTINCT, UPDATE	Standard	Yes
27	Ordered scalar subquery breaks groupby lowering	fixed (#7794)	LIMIT/OFFSET, scalar subquery, UPDATE	Dialect	Yes
28	BETWEEN over nested groups breaks project lowering	fixed (#7795)	BETWEEN, nested subquery	Standard	Yes
29	Nested SELECT expression misses alias	fixed (#7796)	NULL predicate, scalar subquery	Dialect	Yes
30	ORDER BY subquery leaves expression uninitialized	fixed (#7797)	ORDER BY LIMIT, subquery	Standard	Yes
31	ORDER BY boolean subquery breaks projections	fixed (#7800)	NULLS LAST, boolean ORDER BY	Dialect	Yes
32	Correlated UPDATE subquery breaks rel visitor	fixed (#7801)	LIMIT/OFFSET, correlated subquery, UPDATE	Standard	Yes
Virtuoso					
33	Large VARCHAR insert overflows row fill logic	fixed (#1368)	REPEAT, VARCHAR	Dialect	No
34	DROP COLUMN crashes on missing table	fixed (#1370)	ALTER TABLE DROP COLUMN	Dialect	Yes
35	ORDER BY mod(*) crashes execution	fixed (#1371)	mod, ORDER BY	Dialect	Yes
36	Type-mismatched subqueries trigger segmentation fault	fixed (#1372)	NATURAL JOIN, BIT_XOR, subquery	Dialect	No
37	Nonexistent CHECK subquery crashes sqlintrp	fixed (#1373)	CHECK, subquery	Dialect	Yes
38	Bound numeric vector execution crashes	fixed (#1374)	VAR_SAMP, STDDEV_POP, LEAD	Dialect	No
39	Unpositioned cursor crashes UPDATE path	fixed (#1375)	cursor update, UPDATE	Dialect	No
40	Aggregate subquery creation triggers assertion	fixed (#1376)	CASE WHEN, aggregate subquery	Dialect	Yes
41	Missing DFE_FALSE check crashes optimizer	fixed (#1378)	DFE_FALSE, LAG, NATURAL JOIN	Dialect	No
42	Optimizer misses prefix check and dereferences NULL	fixed (#1379)	PERCENT_RANK, NTH_VALUE, NATURAL JOIN	Dialect	No
43	key_estimate misses column existence check	fixed (#1381)	key_estimate, subquery	Dialect	No
44	NULL key_name crashes dsig_template_ext	fixed (#1382)	dsig_template_ext, NULL key_name	Dialect	No
45	FULL OUTER plus NATURAL JOIN crashes jp_add	waiting (#1385)	FULL OUTER JOIN, DENSE_RANK, NATURAL JOIN	Dialect	No
46	Complex view insertion crashes sqlc_insert_view	waiting (#1388)	sqlc_insert_view, VAR_SAMP, BIT_XOR	Dialect	No
47	DECIMAL CHECK update crashes num_divide	waiting (#1389)	DECIMAL, CHECK, UPDATE	Dialect	No
48	TIMESTAMP subquery crashes dvcmp comparison	waiting (#1390)	NTILE, NVARCHAR, TIMESTAMP	Dialect	No
49	Aggregated view query crashes Virtuoso execution	waiting (#1391)	LAST_VALUE, BIT_XOR, VIEW	Dialect	No
50	Correlated CASE view creation crashes optimizer	waiting (#1392)	correlated subquery, CASE expression, EXISTS	Dialect	Yes
51	Dropped-table view update crashes execution	waiting (#1393)	PERCENT_RANK, LAST_VALUE, dropped table	Dialect	No
52	DECIMAL operations cause double free	waiting (#1394)	CREATE TRIGGER, DECIMAL, double free	Dialect	Yes
DuckDB					
53	CHECK columns missing during UPDATE chunking	fixed (#20199)	CHECK, UPDATE	Dialect	Yes
54	OVER subquery breaks plan serialization	fixed (#20263)	OVER, to_centuries	Standard	No
55	date_trunc statistics propagation causes internal error	fixed (#20371)	date_trunc, statistics propagation	Standard	No
56	Repeated-column UNIQUE INDEX triggers UB	fixed (#20418)	UNIQUE INDEX, ART prefix	Dialect	No
57	ON CONFLICT references nonexistent column	fixed (#20476)	ON CONFLICT DO NOTHING	Standard	Yes
58	LAG default mismatch causes ASan crash	fixed (#20615)	LAG, OVER	Dialect	Yes
59	TRY with correlated HAVING triggers internal error	fixed (#20616)	TRY(), correlated HAVING, GROUP BY	Standard	No
60	LEAD with COALESCE breaks PhysicalType sizing	fixed (#20481)	COALESCE, LEAD, GROUP BY	Dialect	Yes
ClickHouse					
61	Negative toInt8 crashes randomStringUTF8	fixed (#93890)	randomStringUTF8, toInt8	Dialect	No

Note. Class indicates whether the key bug-triggering features are common SQL features (*Standard*) or DBMS-specific/rare features (*Dialect*). Seed Cov. indicates whether the key bug-triggering features are covered by existing seed corpora, including SQUIRREL's default seeds and official unit/regression tests.

ClickHouse, respectively. Compared with official unit/regression tests where available, SmartFuzz further introduces 100, 53, and 976 additional features on MonetDB, DuckDB, and ClickHouse. More detailed analysis is available at [62].

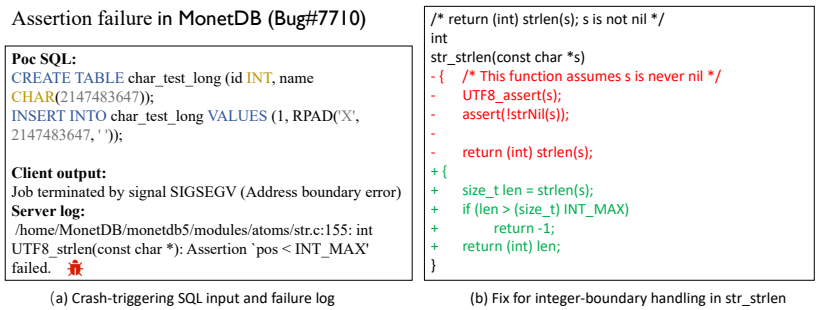


Fig. 7. **Case Study 1: Assertion Failure in MonetDB.**

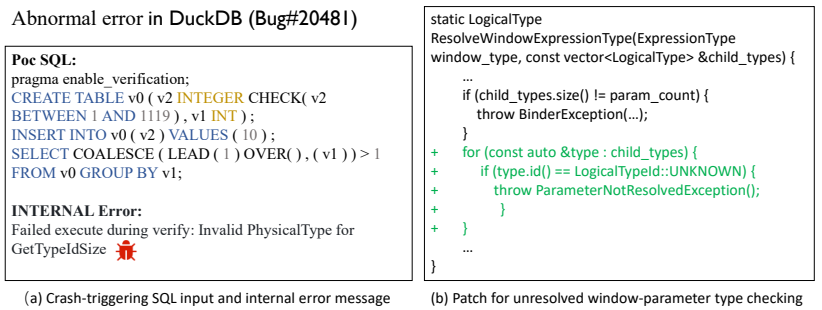


Fig. 8. **Case Study 2: Abnormal Error in DuckDB.**

Case Study 1: Assertion Failure in MonetDB. As shown in Figure 7, this bug is triggered by an RPAD-generated boundary-length string inserted into a "CHAR(2147483647)" column. Specifically, "RPAD('X', 2147483647, ' ')" constructs a string whose length reaches the "INT_MAX" boundary, which is accepted by the SQL layer but not safely handled by MonetDB internally. During execution, MonetDB uses int to represent string length and character positions, and the string-processing path eventually reaches the assertion "pos < INT_MAX" in "UTF8_strlen". The root cause is therefore a mismatch between the externally allowed maximum string length and the internal int-based representation used for length computation. To fix the bug, MonetDB replaces the unsafe int-based handling with a safer size_t-based length check and returns an error when the computed length exceeds "INT_MAX". This case highlights the strength of SmartFuzz in synthesizing semantically valid yet boundary-stressing inputs that combine non-trivial SQL features, such as "RPAD" and extreme "CHAR" lengths.

Case Study 2: Abnormal Error in DuckDB. As shown in Figure 8, this bug is triggered by a query that combines "LEAD(1) OVER()" with "COALESCE" under verification mode. Although the query passes the earlier parsing stages, DuckDB eventually reports an internal error during execution verification. The root cause is that the window-expression type resolver fails to reject an unresolved child type, leaving one parameter as "UNKNOWN" and allowing it to propagate into later physical-type computation. As a result, the execution pipeline attempts to derive a "PhysicalType" from an invalid or uninitialized logical type, which leads to the abnormal internal failure. To fix the bug, DuckDB adds an explicit validation step in "ResolveWindowExpressionType" to check whether any child type remains "UNKNOWN". If such a case is detected, the system now throws "ParameterNotResolvedException" early, preventing invalid types from reaching the physical planning stage. This case further highlights the importance of feature composition, as the bug only emerges when multiple non-trivial constructs, including window functions, "COALESCE", and grouping semantics, are combined into a single executable query.

4.2 Comparison with Existing Seed Selection Strategies

4.2.1 Setup. RQ2 aims to evaluate how different seed selection strategies affect the effectiveness of mutation-based DBMS fuzzing. We implement all configurations on top of the same mutation-based fuzzer, Squirrel, and distinguish them by their seed sources. Specifically, we compare the following six seed selection strategies:

- **Squirrel-*Default***: using the built-in seed corpus shipped with Squirrel.
- **Squirrel-*LocalTests***: using official unit and regression test suites from the target DBMS as initial seeds. Since Virtuoso does not provide comparable official unit/regression test suites, this baseline is not available for Virtuoso.
- **Sedar-*Squirrel***: using cross-DBMS transferred test cases following Sedar [24]. Since the original implementation of Sedar is not publicly available, we re-implement its seed transfer strategy based on the description provided in the paper.
- **SmartFuzz^{NoDoc}-*Squirrel***: removing documentation features; seeds are synthesized *only* from crash-derived features extracted by SmartFuzz.
- **SmartFuzz^{NoCrash}-*Squirrel***: removing crash-derived features; seeds are synthesized *only* from documentation-derived features extracted by SmartFuzz.
- **SmartFuzz-*Squirrel***: using the full SmartFuzz pipeline combining documentation features, crash-derived features, and LLM-driven synthesis.

We follow the same fuzzing environment as in RQ1. For each DBMS, we run fuzzing for 24 hours and record both the number of detected bugs and the achieved coverage. Following Squirrel [78], we repeat each configuration five times for every target DBMS and report the average results to mitigate randomness and improve reliability. This experiment isolates the impact of seed selection and examines whether SmartFuzz can provide higher-quality seeds compared to existing practices.

4.2.2 Results. **Code Coverage.** Figure 9 reports the branch coverage achieved by different seed selection strategies on MonetDB, Virtuoso, DuckDB, and ClickHouse over 24 hours. Overall, SmartFuzz-*Squirrel* consistently achieves the highest branch coverage across all four DBMSs, substantially outperforming fuzzer configurations that rely on default seeds, local test suites, or cross-DBMS seed transfer. Across all four DBMSs, the branch-coverage improvements of SmartFuzz-*Squirrel* over each baseline are statistically significant ($p \leq 0.05$). Compared with Squirrel-*Default*, SmartFuzz-*Squirrel* improves branch coverage by 260.0% on MonetDB, 211.1% on Virtuoso, 256.0% on DuckDB, and 1250.0% on ClickHouse. This highlights the severe limitation of relying solely on fuzzer-shipped default seeds, which cover only a small fraction of executable DBMS logic. When compared against Squirrel-*LocalTests*, which already leverage developer-maintained unit and regression tests, SmartFuzz-*Squirrel* still increases branch coverage by 50.0% on MonetDB, 47.4% on Virtuoso, 27.1% on DuckDB, and 134.8% on ClickHouse. SmartFuzz-*Squirrel* also consistently outperforms Sedar-*Squirrel*, with coverage improvements of 38.5% on MonetDB, 51.4% on Virtuoso, 50.8% on DuckDB, and 300.0% on ClickHouse. This suggests that even high-quality, DBMS-native test suites leave substantial portions of the execution space unexplored during fuzzing. Notably, the coverage curves of SmartFuzz-*Squirrel* rise rapidly during the early fuzzing stage and continue to grow steadily, suggesting that the synthesized seeds enable the fuzzer to quickly reach and explore deep execution paths.

The observed improvements can be attributed to two main factors. First, documentation-feature-driven synthesis injects DBMS-specific SQL constructs that are often absent from default or transferred seed corpora, allowing the fuzzer to exercise execution paths related to rarely tested features. Second, crash-derived feature reuse biases seed generation toward bug-prone execution states, increasing the likelihood that mutations traverse complex and security-critical code regions. In contrast, default seeds and local test cases are primarily designed for functional validation and

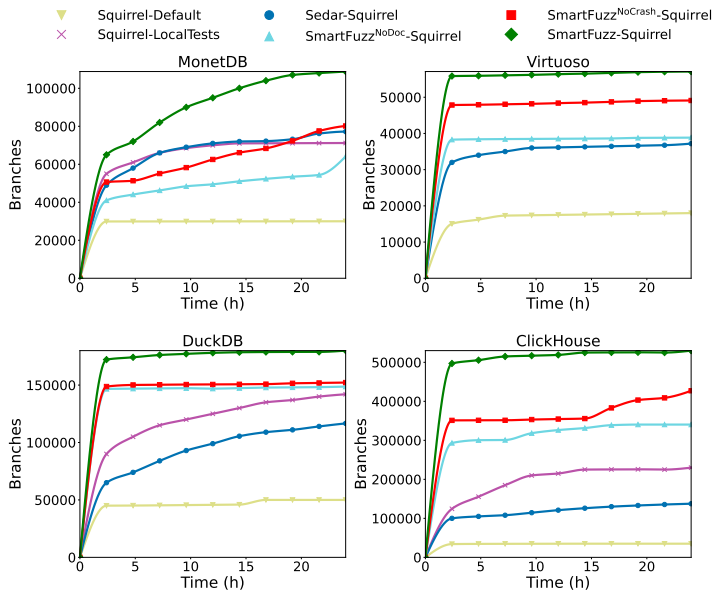


Fig. 9. **The branch coverage of different seed selection strategies on MonetDB, Virtuoso, DuckDB, and ClickHouse.**

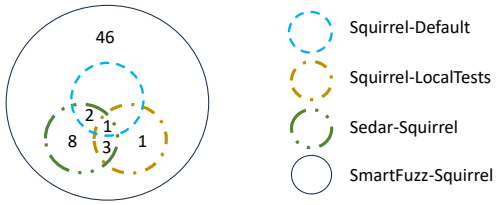


Fig. 10. **Relation of bugs detected by SmartFuzz-*Squirrel*, Squirrel-*Default*, Squirrel-*LocalTests* and Sedar-*Squirrel*.**

tend to concentrate on common usage patterns, which limits their coverage potential during fuzzing. The ablation results further confirm the complementarity of the two feature sources. SmartFuzz^{NoDoc}-*Squirrel* and SmartFuzz^{NoCrash}-*Squirrel* both outperform traditional seed selection strategies but consistently lag behind the full SmartFuzz-*Squirrel* configuration. This indicates that relying solely on documentation-derived or crash-derived features is insufficient to maximize coverage; instead, combining both enables the fuzzer to generate more diverse, semantically valid, and exploration-effective SQL inputs.

Bugs. Figure 10 illustrates the relation among bugs detected by different seed selection strategies. We observe that Squirrel-*Default*, Squirrel-*LocalTests*, and Sedar-*Squirrel* each discover only a small number of bugs, indicating that these baselines explore relatively shallow paths. In contrast, SmartFuzz-*Squirrel* subsumes all bugs detected by the three baselines and further discovers 46 additional bugs that are missed by all other configurations. This result demonstrates that synthesizing target-specific seeds guided by documentation-derived and crash-derived features enables SmartFuzz to expose a substantially broader and deeper set of DBMS bugs than existing seed selection strategies.

Table 6. **Contribution of Crash Analyzer.**

DBMS	Branch Coverage			Bugs	
	SmartFuzz-NoCA	SmartFuzz-CA	Improvement	SmartFuzz-NoCA	SmartFuzz-CA
MonetDB	59.4K	64.1K	+7.9%	11	18
Virtuoso	33.2K	38.0K	+14.5%	5	11
DuckDB	110.5K	143.2K	+29.6%	2	8
ClickHouse	315K	340K	+7.9%	0	0

Notes. SmartFuzz-NoCA: directly reuses translated crash-inducing SQL statements as seeds. SmartFuzz-CA: extracts crash features and synthesizes seeds via feature recomposition.

4.3 Contribution of Crash Analyzer

4.3.1 Setup. We isolate the contribution of SmartFuzz's Crash Analyzer under a crash-only setting. We compare two variants that both rely solely on historical crash sources: (i) **SmartFuzz-NoCA**, which directly uses translated historical crash-inducing SQL statements as initial seeds; and (ii) **SmartFuzz-CA**, which enables Crash Analyzer to extract fine-grained crash features and synthesize seeds by recombining these features (while disabling documentation-derived features). This ablation evaluates whether "feature extraction + recomposition" is more effective than directly replaying translated crash statements. We run each configuration for 24 hours and report (i) branch coverage and (ii) the number of discovered bugs.

4.3.2 Results. Table 6 summarizes the crash-only ablation results. Enabling Crash Analyzer (SmartFuzz-CA) consistently improves branch coverage on all four DBMSs. Specifically, SmartFuzz-CA increases branch coverage by 7.9% on MonetDB, 14.5% on Virtuoso, 29.6% on DuckDB, and 7.9% on ClickHouse. Bug discovery also benefits substantially: MonetDB increases from 11 to 18 bugs (+63.6%), Virtuoso from 5 to 11 (+120.0%), and DuckDB from 2 to 8 (+300.0%). Overall, these results indicate that Crash Analyzer decomposes crashes into reusable, fine-grained features and synthesizes diverse yet executable seeds via feature recomposition, thereby broadening state-space exploration and improving bug discovery even without documentation-derived guidance.

5 Discussion

Seed Generation Validity Across LLMs. To evaluate the effectiveness of our Prompt Composer design and the generalizability of SmartFuzz across diverse LLMs, we conduct a focused study on seed generation validity. While RQ1–RQ2 assess the end-to-end effectiveness of SmartFuzz in bug discovery and coverage, this subsection isolates the component-level validity of the seed synthesis stage, measuring whether the Prompt Composer can reliably guide different LLMs to produce syntactically and semantically correct SQL seeds. For each LLM, we synthesize 500 candidate SQL seeds for each of the four DBMSs (MonetDB, Virtuoso, DuckDB, and ClickHouse), using the same feature selection strategy as in our main evaluation. All models use the identical prompt template described in section 3.3. We measure two metrics: syntax correctness (parsability by the target DBMS) and semantic correctness (successful execution without runtime errors).

Table 7 summarizes the results across nine LLMs, including three closed-source models and six open-source variants from two series (Llama-3.1 and Qwen-3.5). The results show that the Prompt Composer generalizes well across different LLMs, consistently producing a high proportion of syntactically and semantically valid SQL seeds on all four DBMSs. Closed-source frontier models achieve the strongest performance overall, with GPT-5.2 and Gemini-2.5-Pro both maintaining above 90% syntax and semantic correctness across all systems, indicating that stronger reasoning and instruction-following capabilities directly benefit seed synthesis. Although GPT-4o-mini performs slightly below the strongest models, it still achieves high validity on all four DBMSs, suggesting that our main evaluation uses a practical model with a favorable cost–effectiveness trade-off. Among

Table 7. **Semantic and Syntax Correctness of Seed Generation Across Different LLMs and DBMSs.**

Model	MonetDB		Virtuoso		DuckDB		ClickHouse	
	Syntax %	Semantics %	Syntax %	Semantics %	Syntax %	Semantics %	Syntax %	Semantics %
GPT-5.2 [50]	94.5%	92.1%	93.8%	91.5%	93.5%	91.2%	92.8%	90.5%
Gemini-2.5-Pro [17]	94.2%	91.8%	93.5%	91.2%	93.2%	90.8%	92.5%	90.1%
GPT-4o-mini [49] (Main)	90.5%	87.2%	89.8%	86.5%	89.1%	85.8%	88.2%	84.5%
Llama-3.1-8B [6]	73.2%	66.5%	72.1%	64.2%	68.1%	62.0%	67.0%	61.0%
Llama-3.1-70B [5]	85.0%	83.5%	84.5%	81.8%	84.2%	79.0%	83.0%	78.0%
Llama-3.1-405B [4]	87.0%	86.8%	85.9%	85.1%	87.3%	82.4%	86.0%	82.0%
Qwen-3.5-7B [68]	72.0%	61.8%	70.0%	59.5%	63.2%	57.5%	62.0%	55.0%
Qwen-3.5-32B [66]	85.3%	80.2%	83.5%	78.5%	81.1%	75.8%	80.7%	75.6%
Qwen-3.5-72B [67]	88.4%	84.9%	87.1%	83.2%	85.7%	80.6%	84.2%	79.8%

Table 8. **Comparison with Generation-Based Fuzzers SQLancer and SQLancer++.**

DBMS	Baseline	Baseline Cov.	SmartFuzz Cov.	Cov. Gain	Baseline Bugs	SmartFuzz Bugs
MonetDB	SQLancer++	29,251	108,823	272.0%	4	32
Virtuoso	SQLancer++	25,687	57,101	122.3%	1	20
DuckDB	SQLancer	31,789	180,000	466.2%	0	8
ClickHouse	SQLancer	46,812	530,003	1,032.2%	0	1

Note. Cov. denotes branch coverage. Cov. Gain is computed as (SmartFuzz Cov. − Baseline Cov.) /Baseline Cov.. Bugs denotes the number of unique bugs discovered within the same fuzzing budget.

open-source models, validity improves steadily with model size in both the Llama-3.1 and Qwen-3.5 series. Overall, these results indicate that the seed synthesis stage of SmartFuzz is robust across diverse LLMs, while stronger models can further raise the upper bound of seed validity.

Comparison with Generation-Based Fuzzers. To further understand how SmartFuzz compares with generation-based fuzzers, we compare it against two state-of-the-art generation-based fuzzers, SQLancer [63] and SQLancer++ [81], under the same experimental setup as Section 4.2. SQLancer++ is an extensible variant of SQLancer designed to facilitate porting generation-based testing and test oracles to additional DBMSs. For DBMSs natively supported by SQLancer, i.e., DuckDB and ClickHouse, we directly use SQLancer as the baseline; for MonetDB and Virtuoso, which are not natively supported by SQLancer, we use SQLancer++ and add thin DBMS-specific support as needed. As shown in Table 8, SmartFuzz improves branch coverage by 272.0%, 122.3%, 466.2%, and 1,032.2% on MonetDB, Virtuoso, DuckDB, and ClickHouse, respectively. The generation-based baselines find 5 bugs in total, while SmartFuzz finds 61 bugs under the same 24-hour budget. These results show that SmartFuzz is complementary to generation-based DBMS testing and provides practical benefits for both coverage expansion and bug discovery.

This difference can be explained by the different exploration mechanisms of the two paradigms. Generation-based fuzzers construct SQL from grammars or ASTs, but usually lack runtime feedback during input generation, so their exploration is often unguided and bounded by manually encoded grammars and generation rules. Even if their grammars contain features mined by SmartFuzz, this only means that such features may appear in generated queries; it does not ensure diverse placements, feature combinations, or executions along bug-prone paths. Moreover, extending generation-based fuzzers to support new DBMS features is labor-intensive, as it requires manually modeling grammar rules, schema dependencies, value constraints, and DBMS-specific semantics. For example, supporting selected SQLite features in SQLancer [63] requires 10,020 lines of code [81]. In contrast, mutation-based fuzzers use coverage/crash feedback to retain promising seeds and explore different contexts around existing features through insert/delete/replace mutations. SmartFuzz complements this process by supplying LLM-generated feature-rich seeds together with compatibility-aware mutation, enabling deeper testing with lower manual feature-modeling effort.

Table 9. **Comparison with Fuzz4All.**

DBMS	Baseline	Baseline Cov.	SmartFuzz Cov.	Cov. Gain	Baseline Bugs	SmartFuzz Bugs
MonetDB	Fuzz4All	37,841	108,823	187.6%	2	32
Virtuoso	Fuzz4All	56,087	57,101	1.8%	0	20
DuckDB	Fuzz4All	51,725	180,000	248.0%	0	8
ClickHouse	Fuzz4All	175,589	530,003	201.8%	0	1

Note. Cov. denotes branch coverage. Cov. Gain is computed as (SmartFuzz Cov. − Baseline Cov.) / Baseline Cov.. Bugs denotes the number of unique bugs discovered within the same fuzzing budget.

Comparison with AI-Based Fuzzers. We further compare SmartFuzz with Fuzz4All [75], a representative general AI-based fuzzing framework, under the same experimental setup as Section 4.2. We select Fuzz4All because it can be adapted to generate SQL programs with a lightweight DBMS-specific harness. Other AI-based fuzzers are less suitable as direct baselines: FuzzGPT [18] targets deep learning libraries rather than DBMSs, while ShQveL's [80] implementation is not publicly available. For fairness, we adapt Fuzz4All to generate SQL programs for each target DBMS and execute them using the same validation, crash collection, deduplication, and coverage measurement pipeline as SmartFuzz. As shown in Table 9, SmartFuzz improves branch coverage by 187.6%, 1.8%, 248.0%, and 201.8% on MonetDB, Virtuoso, DuckDB, and ClickHouse, respectively. In terms of bug discovery, Fuzz4All finds 2 bugs in total, while SmartFuzz finds 61 bugs under the same 24-hour budget. These results show the practical benefit of DBMS-specific and bug-relevant seed guidance. They also suggest that, in our DBMS fuzzing setting, directly prompting an LLM to generate SQL inputs, as in Fuzz4All, is less effective than using structured guidance from documentation-derived and crash-derived features. Unlike general AI-based fuzzers that rely on LLMs for direct input generation, SmartFuzz uses LLMs only to synthesize initial seeds from documentation-derived and crash-derived features, and then relies on compatibility-aware coverage-guided mutation to explore nearby execution states. Therefore, SmartFuzz represents a different and complementary design point to general AI-based fuzzing frameworks, with better coverage and bug-finding effectiveness in our DBMS fuzzing setting.

Contribution of Coverage-Guided Mutation. In SmartFuzz, we use two notions of coverage. Feature coverage is used only during seed synthesis to avoid repeatedly selecting the same documented or crash-derived features for LLM generation. In contrast, runtime code coverage, is used by the downstream mutation-based fuzzer to guide seed mutation and retain coverage-improving test cases. This runtime feedback is orthogonal to feature coverage and is essential for exploring the execution space around LLM-generated seeds. LLM generation is effective at injecting diverse DBMS-specific and bug-relevant features into the initial seed pool, while coverage-guided mutation efficiently explores nearby execution contexts around these seeds. With runtime feedback, the fuzzer can retain promising seeds and repeatedly mutate them through insertion, deletion, and replacement, thereby exercising different syntactic contexts and execution paths. This avoids the high cost and relatively unguided nature of continuously invoking the LLM to generate new SQL programs from scratch. To further isolate the contribution of runtime coverage-guided mutation, we add an *LLM-only* baseline. This baseline uses the same feature extraction, feature selection, LLM-driven synthesis, and validity filtering as SmartFuzz, but disables runtime coverage-guided mutation and continuously synthesizes new SQL programs under the same 24-hour budget. As shown in Table 10, SmartFuzz improves branch coverage over this *LLM-only* baseline by 88.0%, 1.3%, 254.5%, and 340.5% on MonetDB, Virtuoso, DuckDB, and ClickHouse, respectively, and discovers 57 additional bugs. These results show that LLM-based seed synthesis and runtime coverage-guided mutation play complementary roles: the former introduces diverse and bug-relevant features, while the latter efficiently explores their surrounding execution space.

Table 10. **Contribution of Coverage-Guided Mutation over *LLM-only* Generation.**

DBMS	Baseline	Baseline Cov.	SmartFuzz Cov.	Cov. Gain	Baseline Bugs	SmartFuzz Bugs
MonetDB	*LLM-only*	57,873	108,823	88.0%	0	32
Virtuoso	*LLM-only*	56,389	57,101	1.3%	4	20
DuckDB	*LLM-only*	50,782	180,000	254.5%	0	8
ClickHouse	*LLM-only*	120,326	530,003	340.5%	0	1

Note. Cov. denotes branch coverage. Cov. Gain is computed as (SmartFuzz Cov. − Baseline Cov.) / Baseline Cov.. Bugs denotes the number of unique bugs discovered within the same fuzzing budget.

Threats to Validity. Internal validity concerns primarily involve experimental fairness. Since SEDAR [24] is not open-source, we reimplemented its compatibility-aware mutation module based on the paper description; to mitigate implementation discrepancies, we manually wrote test cases to verify the correctness of key mutation behaviors against the reported semantics. Another internal validity threat concerns the interpretation of comparisons across fuzzing paradigms. Although our comparison with SQLANCER/SQLANCER++ uses the same fuzzing budget, it is not a fully apples-to-apples component comparison. SmartFuzz uses an LLM to synthesize feature-rich initial seeds, while SQLANCER/SQLANCER++ rely on manually encoded grammars and do not use LLMs to update their generators or maximize syntax coverage. Thus, the results should be interpreted as demonstrating the practical benefit of combining LLM-guided seed synthesis with coverage-guided mutation, rather than proving that mutation-based fuzzing is universally superior to generation-based fuzzing. Similarly, LLM-based fuzzers such as FUZZ4ALL represent another design point with different trade-offs in generation cost, throughput, validity filtering, and feedback usage. Therefore, we view SmartFuzz as complementary to generation-based and LLM-based fuzzing approaches. External validity threats relate to generalizability across LLMs and DBMSs. First, we employ GPT-4o-mini as the underlying synthesis model; model-specific behaviors (e.g., prompt sensitivity, reasoning patterns) may influence seed quality. To assess generalizability, we conducted preliminary evaluations on alternative models (e.g., Llama and Qwen) as discussed in the above subsection, observing consistent trends in seed generation performance. Second, our evaluation is limited to four open-source relational DBMSs. Results may not fully generalize to proprietary systems or NoSQL databases due to dialect and execution logic variations. Although our compatibility-aware module mitigates dialect mismatches for relational SQL, adapting to non-relational query languages would require extending the parsing and mutation infrastructure, as SQUIRREL's parser and mutator are designed specifically for relational DBMSs.

6 Related Work

6.1 DBMS Fuzzing

Mutation-based fuzzers create new SQL test cases by mutating existing inputs under coverage guidance. General-purpose mutation engines such as AFL [29], LIBFUZZER [44], and HONGGFUZZ [28] are widely used (for example, within OSS-Fuzz) to evaluate in-memory DBMSs like SQLite. To produce more meaningful mutations, these fuzzers commonly maintain dictionaries of SQL keywords and idioms to steer mutation operators, yet naively mutated queries still suffer from abundant syntactic or semantic invalidity. Recent research therefore integrates SQL grammar and semantic modeling into the mutation loop. SQUIRREL [78] introduces an intermediate representation (IR) derived from the AST and performs IR-level mutations that respect object dependencies to preserve semantic correctness. RATEL [70] improves feedback precision by employing bijective block mapping for finer-grained coverage guidance. LEGO [37] mines type-affinity patterns from existing cases and synthesizes inputs with realistic SQL type sequences. UNICORN [74] presents a hybrid synthesis approach tailored to time-series DBMSs by injecting time-series elements into queries. SEDAR [24]

transfers test cases from other popular DBMSs. Griffin [25] proposes a grammar-free mutation strategy that reshuffles statements from real queries and applies metadata-guided semantic fixes.

Generation-based fuzzers create SQL test cases according to predefined grammars or generation rules. For example, SQLsmith [61] models SQL syntax as an abstract syntax tree and systematically enumerates queries, flagging bugs when generated inputs cause server disconnections. SQLancer [58–60] adopts multiple test oracles to detect semantic and logic errors; its NOREC oracle, for instance, constructs queries containing WHERE and JOIN clauses, derives an equivalent form by moving predicates, and compares results to reveal inconsistencies. Mozi [38] leverages configuration-aware equivalence transformations to detect both logic and performance faults. Apollo [34] builds on SQLsmith as the query generator and performs cross-version regression comparisons to identify performance anomalies across DBMS releases.

SmartFuzz generates high-quality initial seeds that can be directly consumed by popular DBMS fuzzing tools. Unlike generation-based approaches that rely on predefined grammars or hand-crafted rules and therefore tend to cover only a limited subset of DBMS features, and unlike mutation-based approaches that depend heavily on manually collected seeds or regression suites, SmartFuzz synthesizes seeds by combining documentation-derived features with crash-reuse signals. This hybrid synthesis yields seeds that are compliant with the target DBMS and enriched with crash-relevant traits. As a result, SmartFuzz can be used in two complementary ways: it can act as a generation engine that emits DBMS-aware test cases tailored to specific systems, or it can serve as a seed provider that supplies diverse, bug-focused initial inputs for mutation-based fuzzers, thereby broadening their exploration and improving bug discovery.

6.2 Initial Seed Generation

Prior work has explored various strategies to produce high-quality initial seeds for mutation-based fuzzing [42, 54, 69, 76]. Techniques range from symbolic execution (as in SAFL) that synthesizes seeds to steer exploration [69], to approaches that identify program input-validity checks and generate inputs consistent with those checks (SLF) [76], and to constraint-solving methods that treat input fields as string variables and solve for satisfying assignments (TensileFuzz) [42]. While these methods are effective for many general-purpose fuzzing targets, they face difficulties when applied to DBMSs because SQL imposes far more intricate syntactic and semantic constraints than the input formats these techniques were designed for. In the DBMS fuzzing domain, recent work begins to address this gap: Sedar [24] transfers test cases from other widely used DBMSs to bootstrap mutation-based testing for a target DBMS, thereby leveraging cross-DBMS similarities to obtain stronger seeds.

Existing seed-generation techniques typically focus on domain-specific or text-oriented inputs, and therefore struggle to handle SQL's complex syntactic and semantic dependencies. SmartFuzz is the first work to leverage LLMs for synthesizing DBMS-aware initial seeds: it systematically combines documentation-derived features with crash-derived signals via LLM-driven feature composition to produce seeds that are both syntactically and semantically compliant and enriched with crash-relevant patterns.

6.3 LLM-assisted Fuzzing

Recent advances in LLMs, such as OpenAI's GPT-4 Turbo, have spurred new research that applies LLMs to improve fuzzing. Inspired by LLMs' strong abilities in text generation and understanding, researchers leverage their knowledge learned from massive corpora to generate code and structured inputs across domains, thereby improving fuzzing efficiency and coverage. Systems such as Codamosa [36], TitanFuzz [18], FuzzGPT [19], CovRL [21], and Fuzz4All [75] demonstrate the effectiveness of LLMs for generating new fuzz targets. Beyond test-case synthesis, LLMs

have been explored as intelligent mutation engines and as automatic driver generators: PromptFuzz [46] shows that LLMs can produce fuzz drivers for frameworks like libFuzzer [43], while CHATFUZZ [32] leverages prompts to instruct an LLM to create multiple valid variants of an existing seed so that variants can pass initial parsing and exercise deeper program logic.

Inspired by these prior works, SmartFuzz also leverages LLM capabilities to specifically explore seed synthesis for DBMS fuzzing. However, SmartFuzz differs from general AI-based fuzzers in two key aspects. First, unlike many AI-based fuzzers such as FuzzGPT [19] and Fuzz4All [75], which use LLMs throughout the fuzzing loop or directly generate test inputs, SmartFuzz uses LLMs only for initial seed generation. After the seed pool is synthesized, SmartFuzz relies on existing mutation-based fuzzers for efficient coverage-guided exploration, reducing repeated LLM invocation cost and improving fuzzing throughput. To make this integration practical, SmartFuzz introduces compatibility-aware mutation to preserve LLM-generated dialect-specific SQL fragments that are unsupported by existing mutation engines. Second, SmartFuzz provides DBMS-specific guidance to the LLM. Instead of relying only on general prompts or examples, SmartFuzz extracts documentation-derived features and crash-derived features, and injects them into prompts to provide target-dialect knowledge and bug-prone patterns. These feature-guided seeds increase the diversity and bug-triggering potential of the initial seed pool, enabling mutation-based fuzzers to explore richer DBMS execution states.

6.4 Fuzzing with Historical Bugs

Fuzzing with historical bugs has been extensively studied. One of the pioneering techniques is LangFuzz [31], which parses historical test cases known to trigger program misbehavior and extracts individual code fragments. These fragments are then partially replaced into newly generated code to synthesize novel test inputs. Researchers have also mined fuzzer configurations (e.g., grammar or statement occurrence probabilities) from historical bug-triggering programs to enable more diverse input generation [9, 65]. More recently, JavaTailor [77] leverages "code ingredients" extracted from historical bugs to synthesize new input programs for Java Virtual Machine (JVM) fuzzing. Besides such generation-based or hybrid approaches, mutation-based fuzzers have also widely utilized prior bug reports or regression tests as high-quality initial seeds for subsequent mutations, targeting domains such as C compilers [35] and SMT solvers [73].

In contrast, SmartFuzz goes beyond directly reusing historical crash-inducing inputs. It extracts fine-grained *crash features* from historical SQL statements that previously triggered DBMS bugs and recombines them through LLM-guided synthesis to generate new seeds. This feature extraction and recomposition enable deeper exploration of latent bugs that simple input reuse may miss. Moreover, due to the complex syntax and strong semantic dependencies inherent to SQL, reusing crash features for DBMS fuzzing poses greater challenges compared to traditional program fuzzing; SmartFuzz overcomes this difficulty by leveraging the reasoning and synthesis capabilities of modern LLMs to automatically compose diverse and executable initial seeds.

7 Conclusion

This paper identified seed selection as a key bottleneck for mutation-based DBMS fuzzing. We presented SmartFuzz, a framework that leverages documentation-derived features and crash-derived traits to synthesize executable, feature-rich SQL seeds using LLMs. These synthesized seeds expand the feature space available to mutation engines and bias fuzzing toward deeper and more bug-prone execution paths. Integrated into existing fuzzing pipelines, SmartFuzz improves both coverage and bug discovery across 4 DBMSs including MonetDB, Virtuoso, DuckDB, ClickHouse, uncovering 61 previously unknown bugs.

Data-Availability Statement

We release our code and data in our public GitHub repository https://github.com/SmartFuzz/SmartFuzz.

Acknowledgements

We sincerely thank the anonymous reviewers and shepherd for their valuable and insightful feedback. We also thank the DBMS developers for triaging and fixing our reported bugs. This research was supported by the Natural Science Foundation of China (Grant No. 62272400), the Fujian Provincial Natural Science Foundation of China (Grant No. 2025J010002), and Xinmiao Project of ZheJiang Provincial Applied Basic Research Program (Grant No. 2026XMZD032). Li Lin began this work while affiliated with Xiamen University. Now he is a Ph.D. student at Zhejiang University. Rongxin Wu is the corresponding author and works as a member of Xiamen Key Laboratory of Intelligent Storage and Computing in Xiamen University.

References

[1] 2022. ANSI Standard. https://www.ansi.org/. Accessed: 2025-10-24.
[2] 2026. Monetdb crash when using field fcuntion #7709. https://github.com/MonetDB/MonetDB/issues/7709. 2026-01-23.
[3] 2026. SQLite ticket d87336c81c (LAG window function bug report). https://www.sqlite.org/src/tktview/d87336c81c. 2026-01-23.
[4] Meta AI. 2024. Llama-3.1-405B. https://huggingface.co/meta-llama/Llama-3.1-405B. Accessed: 2026-01.
[5] Meta AI. 2024. Llama-3.1-70B. https://huggingface.co/meta-llama/Llama-3.1-70B. Accessed: 2026-01.
[6] Meta AI. 2024. Llama-3.1-8B. https://huggingface.co/meta-llama/Llama-3.1-8B. Accessed: 2026-01.
[7] Cyrille Artho. 2011. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer* 13, 3 (2011), 223–246.
[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.
[9] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-guided configuration diversification for compiler test-program generation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 305–316.
[10] Xinyue Chen, Chenglong Wang, and Alvin Cheung. 2020. Testing query execution engines with mutations. In *Proceedings of the workshop on Testing Database Systems*. 1–5.
[11] Yongxin Chen, Zhiyuan Jiang, Chao Zhang, Haoran Xu, Shenglin Xu, Jianping Tang, Zheming Li, Peidai Xie, and Yongjun Wang. 2026. FuzzySQL: Uncovering Hidden Vulnerabilities in DBMS Special Features with LLM-Driven Fuzzing. *arXiv preprint arXiv:2602.19490* (2026).
[12] ClickHouse Inc. 2025. ClickHouse Website. https://clickhouse.com/. Accessed: December 24, 2025.
[13] SQLite Consortium. 2025. SQLite Home Page. https://sqlite.org/index.html. Accessed: 2025-01.
[14] Oracle Corporation. 2025. MySQL Database. https://www.mysql.com/. Accessed: 2025-01.
[15] OceanBase Corporation. 2025. OceanBase Open Source Database. https://www.oceanbase.com/product/opensource. Accessed: 2025-01.
[16] CR7-source. 2026. ASan crash in LAG() with VARCHAR column, large offset, and non-string default value (Streaming-Window) #20615. https://github.com/duckdb/duckdb/issues/20615. Accessed: 2026-01.
[17] Google DeepMind. 2025. Gemini 2.5: A new era of multimodal AI. https://deepmind.google/technologies/gemini/. Accessed: 2026-01.
[18] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv preprint arXiv:2304.02014* (2023).
[19] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*. 1–13.
[20] DuckDB. 2025. DuckDB Website. https://www.duckdb.org/. Accessed: December 24, 2025.
[21] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Covrl: Fuzzing javascript engines with coverage-guided reinforcement learning for llm-based mutation. *arXiv preprint arXiv:2402.12222* (2024).
[22] Usama M Fayyad. 2011. Data science revealed: A data-driven glimpse into the burgeoning new field.

[23] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its nature, scope, limits, and consequences. *Minds and machines* 30, 4 (2020), 681–694.
[24] Jingzhou Fu, Jie Liang, Zhiyong Wu, and Yu Jiang. 2024. Sedar: Obtaining high-quality seeds for dbms fuzzing via cross-dbms sql transfer. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.
[25] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-free dbms fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[26] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. {GREYONE}: Data flow sensitive fuzzing. In *29th USENIX security symposium (USENIX Security 20)*. 2577–2594.
[27] Xiyue Gao, Zhuang Liu, Jiangtao Cui, Hui Li, Hui Zhang, Kewei Wei, and Kankan Zhao. 2023. A comprehensive survey on database management system fuzzing: Techniques, taxonomy and experimental comparison. *arXiv preprint arXiv:2311.06728* (2023).
[28] Google. 2024. honggfuzz: Security-oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz. GitHub repository. Accessed: 2025-10-12.
[29] Google. 2026. *American Fuzzy Lop (AFL)*. https://github.com/google/AFL Accessed: 2025-10-07.
[30] The PostgreSQL Global Development Group. 2025. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/. Accessed: 2025-01.
[31] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. 445–458.
[32] Jie Hu, Qian Zhang, and Heng Yin. 2023. Augmenting greybox fuzzing with generative ai. *arXiv preprint arXiv:2306.06782* (2023).
[33] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2020. Fuzzing error handling code using {Context-Sensitive} software fault injection. In *29th USENIX security symposium (USENIX Security 20)*. 2595–2612.
[34] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment* 13, 1 (2019), 57–70.
[35] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.
[36] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.
[37] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-oriented DBMS fuzzing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 668–681.
[38] Jie Liang, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Chengnian Sun, and Yu Jiang. 2024. Mozi: Discovering dbms bugs via configuration-based equivalent transformation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.
[39] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting logical bugs of {DBMS} with coverage-based guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. 4309–4326.
[40] Li Lin, Zongyin Hao, Chengpeng Wang, Zhuangda Wang, Rongxin Wu, and Gang Fan. 2024. SQLess: Dialect-Agnostic SQL Query Simplification. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 743–754.
[41] Li Lin, Qinglin Zhu, Hongqiao Chen, Zhuangda Wang, Rongxin Wu, and Xiaoheng Xie. 2025. QTRAN: Extending Metamorphic-Oracle Based Logical Bug Detection Techniques for Multiple-DBMS Dialect Support. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 731–752.
[42] Xuwei Liu, Wei You, Zhuo Zhang, and Xiangyu Zhang. 2022. TensileFuzz: facilitating seed input generation in fuzzing via string constraint solving. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 391–403.
[43] LLVM Project. 2024. libFuzzer — a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2025-10-09.
[44] LLVM Project. 2024. LibFuzzer: A library for coverage-guided fuzzing. https://www.llvm.org/docs/LibFuzzer.html. Accessed: 2025-10-12.
[45] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX security symposium (USENIX security 19)*. 1949–1966.
[46] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt fuzzing for fuzz driver generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3793–3807.
[47] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. 142–151.
[48] MonetDB B.V. 2025. MonetDB Website. https://www.monetdb.org. Accessed: December 24, 2025.

[49] OpenAI. 2024. GPT-4o mini Model. https://platform.openai.com/docs/models/gpt-4o-mini. Accessed: 2025-01.
[50] OpenAI. 2025. GPT-5 System Card. https://openai.com/index/gpt-5-system-card/. Accessed: 2026-01.
[51] OpenLink Software. 2025. Virtuoso Open-Source Edition. https://vos.openlinksw.com/owiki/wiki/VOS. Accessed: December 24, 2025.
[52] Oracle Corporation. 2026. MySQL 8.4 Reference Manual: Information Functions. https://dev.mysql.com/doc/refman/8.4/en/information-functions.html Accessed: 2026-06-14.
[53] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340.
[54] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. {MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. 729–743.
[55] PostgreSQL Global Development Group. 2026. PostgreSQL Documentation: System Catalog Information Functions. https://www.postgresql.org/docs/current/functions-info.html Accessed: 2026-06-14.
[56] Theofanis P Raptis, Andrea Passarella, and Marco Conti. 2019. Data management in industry 4.0: State of the art and open challenges. *Ieee Access* 7 (2019), 97052–97093.
[57] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.
[58] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1140–1152.
[59] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.
[60] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 667–682.
[61] Andreas Seltenreich. n.d.. *SQLSmith*. https://github.com/anse1/sqlsmith Accessed: 2025-10-07.
[62] SmartFuzz Authors. [n. d.]. Additional Feature-Level Analysis. https://github.com/SmartFuzz/SmartFuzz/blob/main/Additional%20Feature-level%20Analysis.md. Supplementary material for SmartFuzz. Accessed: 2026-06-14.
[63] SQLancer Developers. [n. d.]. SQLancer. https://github.com/sqlancer/sqlancer. Automated testing to find logic and performance bugs in database systems. Accessed: 2026-06-14.
[64] Michael Stonebraker, Sam Madden, and Pradeep Dubey. 2013. Intel" big data" science and technology center vision and execution plan. *ACM SIGMOD Record* 42, 1 (2013), 44–49.
[65] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*. 849–863.
[66] Qwen Team. 2024. Qwen2.5-32B. https://huggingface.co/Qwen/Qwen2.5-32B. Accessed: 2026-01.
[67] Qwen Team. 2024. Qwen2.5-72B. https://huggingface.co/Qwen/Qwen2.5-72B. Accessed: 2026-01.
[68] Qwen Team. 2024. Qwen2.5-7B. https://huggingface.co/Qwen/Qwen2.5-7B. Accessed: 2026-01.
[69] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 61–64.
[70] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry practice of coverage-guided enterprise-level DBMS fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 328–337.
[71] Jin Wei, Ping Chen, Kangjie Lu, Jun Dai, and Xiaoyan Sun. 2024. SQLaser: Detecting DBMS Logic Bugs with Clause-Guided Fuzzing. *arXiv preprint arXiv:2407.04294* (2024).
[72] Shihao Wen, Peng Jia, Pin Yang, and Chi Hu. 2023. Squill: testing DBMS with correctness feedback and accurate instantiation. *Applied Sciences* 13, 4 (2023), 2519.
[73] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25.
[74] Zhiyong Wu, Jie Liang, Mingzhe Wang, Chijin Zhou, and Yu Jiang. 2022. Unicorn: detect runtime errors in time-series databases with hybrid input synthesis. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 251–262.
[75] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[76] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: Fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 712–723.

[77] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-driven test program synthesis for JVM testing. In *Proceedings of the 44th International Conference on Software Engineering*. 1133–1144.

[78] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970.

[79] Suyang Zhong and Manuel Rigger. 2024. Understanding and Reusing Test Suites Across Database Systems. *Proceedings of the ACM on Management of Data* 2, 6 (2024), 1–26.

[80] Suyang Zhong and Manuel Rigger. 2025. Testing Database Systems with Large Language Model Synthesized Fragments. *arXiv preprint arXiv:2505.02012* (2025).

[81] Suyang Zhong and Manuel Rigger. 2026. Scaling Automated Database System Testing. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1677–1692.

Received 2026-03-17; accepted 2026-08-06