

QUERYHOUSE: Cross-DBMS Differential Testing with LLM and Query Transpilation

Seoyeon Oh¹, Juhyun Song¹, Jaemin Park², Kwanghee Lee³, Minae Hyeon⁴,
Minji Kim⁵, Sehyuk Ka⁶, Gwangwun Jung⁷, Brian Pak⁷, and Insu Yun¹

¹ KAIST

² University of Ulsan

³ Kangwon National University

⁴ Jeju National University

⁵ Sungshin Women’s University

⁶ Korea University, ¹⁻⁶ Republic of Korea

⁷ Theori, Inc.

{seoyeonoh, juhyun167, insuyun}@kaist.ac.kr, pdas04@mail.ulsan.ac.kr,
rhkdgm14328@kangwon.ac.kr, mhyeon0033@stu.jejunu.ac.kr, hzuw33@gmail.com,
dpp0548@korea.ac.kr, {pr0ln, brian}@theori.io

Abstract. Database Management Systems (DBMSs) are foundational components in modern software development, powering nearly every data-driven application. Recent efforts have increasingly focused on detecting logic bugs, beyond memory and performance issues, to ensure not only availability but also operational correctness. However, prior testing methods have targeted specific DBMSs and bug types, limiting comprehensive approaches due to syntactic and semantic discrepancies in SQL dialects across DBMSs.

We propose QUERYHOUSE, a differential testing framework for discovering logic bugs in a cross-DBMS context, leveraging a SQL transpiler and LLMs. Specifically, we employ a customized transpiler to generate adapted queries while resolving SQL discrepancies, and adopt a hybrid mutation strategy that augments query mutation with an LLM to enhance exploration. We evaluate QUERYHOUSE on five widely used DBMSs—Oracle, MySQL, PostgreSQL, SQLite, and MariaDB—and uncovered 11 previously unknown bugs and 7 undocumented inconsistencies. Our experiments confirm that the proposed methods improve query validity and exploration. We believe QUERYHOUSE enables scalable detection of core logic bugs, thereby enhancing the robustness of DBMS software and their applications.

Keywords: Differential Testing · Database Logic Bugs · LLM.

1 INTRODUCTION

We live in a data-driven world; hence, database management systems (DBMSs) are virtually everywhere [22,49,40]. They are among the most crucial components in modern software that handles data [48,10], ranging from everyday applications

like online shopping and banking to complex systems such as military information infrastructures [16,6,37,9]. Given their deep integration with business logic and user data, failures in DBMSs can have severe consequences [5,26,18]. At a minimum, such failures degrade availability through system hangs or malfunctions. More critically, systems may continue to operate but silently fail, potentially leading to data leaks, compromised integrity, or security violations.

To improve DBMS robustness, prior work has developed automated frameworks for bug detection [58,30,46,44,45,52,7,23,57,27]. Beyond memory and performance bugs [47,41,28,33], interest has focused on logic bugs, which produce incorrect results without visible failures [30,27]. A prominent approach is metamorphic testing [46,45,7], which transforms a query Q for a DBMS D into Q' and verifies whether $D(Q)$ and $D(Q')$ satisfy expected relationships. Various oracles, including NoRec [44], Pinolo [23], DQE [52], and CODDTest [57], target different classes of logic bugs. Recently, QTRAN [31] utilized a Large Language Model (LLM) in SQL transpilation to extend them to additional DBMS dialects.

Despite these advances, a fundamental question for metamorphic testing remains: *What if a DBMS implementation of a feature is fundamentally flawed?* For example, a SQL construct may fail entirely in a DBMS regardless of input predicates [17], necessitating completely different detection strategies. The situation is worsened by what we call the *SQL discrepancy problem*. Although standards like ISO/IEC 9075 exist [1,14], DBMS vendors often deviate from them, intentionally or otherwise [12,59]. Consequently, the burden *falls* on the DBMS testing community, forcing testing to be DBMS- and bug-specific.

To overcome these limitations, we propose a scalable approach based on differential testing [36,15,11]. Given n DBMSs $D_1, D_2, \dots, D_n$, a query Q_1 is transformed into $Q_2, \dots, Q_n$, and the results $D_1(Q_1), D_2(Q_2), \dots, D_n(Q_n)$ are checked for equivalence. Any deviation is flagged for further triage. Specifically, discrepancies contradicting official documentation are treated as bugs, while undocumented differences are classified as inconsistencies. Based on this method, we present QUERYHOUSE, a general differential testing framework for logic bug detection in a cross-DBMS context. QUERYHOUSE employs SQL transpilation with refined translation rules for semantic accuracy and combines rule-based and LLM-based mutation to diversify query generation. We implemented and evaluated QUERYHOUSE on five of the most widely used relational DBMSs, identifying 11 bugs and 7 undocumented inconsistencies. Overall, we believe that QUERYHOUSE and our findings will contribute to improving DBMS robustness by offering a new direction for logic bug finding.

Contributions. This paper makes the following contributions.

- We demonstrate the feasibility of differential testing across multiple DBMSs, ensuring syntactic and semantic correctness for each system while preserving query complexity.
- We implement QUERYHOUSE, a novel automatic framework for cross-DBMS differential testing that identifies fundamental logic bugs. QUERYHOUSE leverages a SQL transpiler and LLM-based mutation to address challenges arising

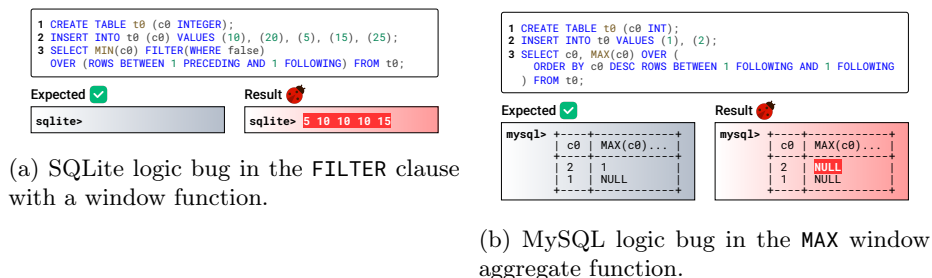


Fig. 1: Queries revealing two logic bugs in the window functions of SQLite and MySQL, discovered using QUERYHOUSE.

from SQL fragmentation and effective query generation. Our artifact is available at GitHub repositories [2,3].

- We evaluate QUERYHOUSE on five widely used relational DBMSs—Oracle, MySQL, PostgreSQL, SQLite, and MariaDB. QUERYHOUSE detected 11 new logic bugs, all confirmed by vendors, and 7 undocumented inconsistencies.

2 BACKGROUND & CHALLENGES

2.1 Motivational Examples

Fig. 1 illustrates two logic bugs [17,56] we discovered in SQLite and MySQL using simple single-column tables, which had evaded detection for 5 and 7 years, respectively. Both bugs stemmed from the window function feature, which applies aggregate functions over a frame surrounding each row. The first bug (Fig. 1a) caused the `FILTER` clause to be entirely ignored, leading to incorrect results whenever some rows should have been excluded. The second bug (Fig. 1b) caused the `MAX` function to incorrectly return `NULL` despite a non-empty window frame. In both bugs, every other DBMS returned the correct result, enabling QUERYHOUSE to immediately flag the anomaly. These are severe functional bugs that under certain conditions silently produce incorrect results without warning, posing a threat to data integrity and business logic, such as filtering policies and rolling maxima calculations.

Why, then, did these bugs remain undetected for so long and evade prior work, despite their relatively simple nature (e.g., easy to reproduce and not stateful)? We argue that they expose limitations in the current testing landscape, particularly in metamorphic testing. First, bugs may be missed when transformed queries lead to *sympathetic failures*—the transformation triggers the *same* defect, causing the results of the “original vs. transformed” queries to remain identical. For example, the `FILTER` clause bug (Fig. 1a) was consistently ignored due to implementation flaws; consequently, rewriting a condition into equivalent predicates would never yield observable differences. Second, existing query-level oracles provide *limited support* for advanced SQL features [27]. Although effective

at identifying bugs in core SQL functionality, most do not offer tailored transformation rules for complex constructs such as window functions, as the complexity of rewrites poses practical challenges. As a result, their transformations are not designed to expose defects within these features. In summary, these blind spots in bug detection persist, underscoring the need for new symmetry-breaking oracles with broader feature reach—a gap that QUERYHOUSE’s cross-DBMS differential testing is designed to address.

2.2 Cross-DBMS Testing Challenges and Approach

Cross-DBMS testing for logic bugs (e.g., §2.1) remains underexplored [11,32], particularly across a large and heterogeneous set of systems (5+), including proprietary DBMSs. We identify three key challenges:

- **C1: Designing Bug Oracles.** Constructing comprehensive oracles for logic bugs is difficult. Existing approaches such as SQLancer [46,44,7] target limited bug classes (e.g., optimization or plan inconsistencies), while SemCont [32] restricts testing to well-standardized features, leaving complex behaviors untested.
- **C2: DBMS-Specific Discrepancies.** Despite SQL standardization [1], DBMSs diverge in syntax and semantics, introducing inconsistencies (e.g., `FETCH` vs. `LIMIT`, or differing `REGEXP_REPLACE` behavior). Generating semantically equivalent queries across systems is therefore challenging [31].
- **C3: Complexity-Validity Tradeoff.** Complex queries are essential for exposing deep bugs, but ensuring validity is difficult, especially across heterogeneous DBMSs. Existing approaches either limit structural diversity [58] or depend on specific oracles [30], restricting generation directions (see §3.3).

To address these challenges, we propose three approaches that underpin QUERYHOUSE:

- **A1: Differential Oracle as Symmetry-Breaker.** We adopt cross-DBMS differential testing (§3.4) to avoid sympathetic failures (§2.1). Given the maturity of DBMSs, at least one system is likely to produce correct outputs. This approach is simple and scalable compared to grammar-specific methods [29].
- **A2: Query Transpilation.** We use automated transpilation (via SQL-Glot [34]) to adapt queries across dialects, mitigating syntactic and semantic discrepancies. We reduce false positives by refining translation rules and excluding unreliable features (§3.2).
- **A3: LLM-Based Mutation.** We introduce LLM-based mutation to promote diverse query exploration. To reduce cost and redundancy, LLMs are invoked only when mutation stalls, based on keyword coverage [32] (§3.3).

3 SYSTEM DESIGN

3.1 Overview of QUERYHOUSE

Fig. 2 presents a high-level overview of QUERYHOUSE. QUERYHOUSE is designed for differential testing across multiple DBMSs by identifying discrepancies in

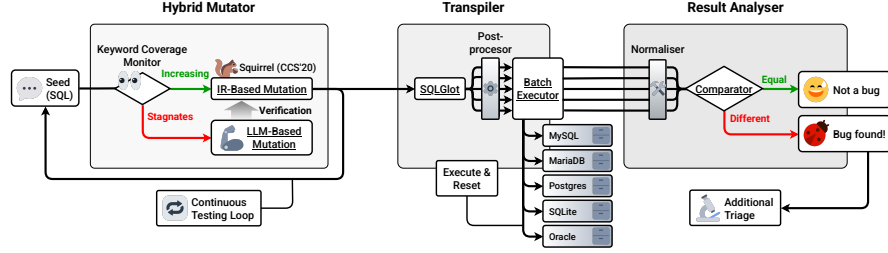


Fig. 2: **Executive workflow of QUERYHOUSE.** The Hybrid Mutator and Transpiler collaborate to generate DBMS-agnostic queries (§3.2). This process leverages a keyword coverage metric, employing LLM-based mutation to overcome coverage stagnation and encourage diverse input generation (§3.3). The Batch Executor runs each query and resets DBMSs to a clean state, while the Result Analyzer compares normalized outputs to identify potential bugs (§3.4).

SQL query execution. It operates in a continuous testing loop, processing input SQL statements that include schema definition, data population, and data retrieval in a single unit.

QUERYHOUSE consists of four modules: Hybrid Mutator, Transpiler, Batch Executor, and Result Analyzer. The Hybrid Mutator performs query mutation using both rule-based and LLM-based methods. LLM-based mutations are selectively applied to promote more diverse exploration when keyword coverage in the rule-based mutation corpus plateaus. The Transpiler then converts mutated queries into the correct syntax for each DBMS, handling syntactic differences while preserving query semantics. The Batch Executor runs these queries on the target DBMSs, grouping them into batches to optimize performance. Finally, the Result Analyzer compares normalized query outputs to detect discrepancies. When different results are found, an additional triage step is conducted to determine if the discrepancy indicates a genuine bug or an unknown inconsistency.

3.2 DBMS-Agnostic Query Generation

The objective of QUERYHOUSE is cross-DBMS differential testing, which requires that queries sent on each target DBMS strictly conform to its SQL dialect. Both the Hybrid Mutator and Transpiler contribute to this goal, and this section describes how they cooperate to ensure query compatibility and adaptability.

Base and Adapted Queries. The Hybrid Mutator generates what we refer to as *base queries*, written in a unified syntax. These base queries are then passed to the Transpiler as input. We chose SQLite syntax for base queries due to its simplicity and broad compatibility with most SQL dialects. Since SQLite supports many common SQL features [55,54], it is highly likely that the base queries utilize features that are also admissible in other DBMSs.

The Transpiler transforms these base queries into forms specific to each target DBMS, producing what we call *adapted queries*. It achieves this by parsing a

base query into a syntax tree and translating it into a concrete query in each dialect, an approach similar to orthogonal work [53] for Object-Relational Mapping (ORM) systems. We tuned the mutator to avoid SQLite-specific elements to further improve the compatibility of these adapted queries. For example, we excluded features such as `WITHOUT ROWID` and `PRAGMA` statements. Similarly, we avoided less commonly supported elements such as window frame definitions using `GROUPS`. This selective approach enhances query versatility and reduces potential compatibility issues when paired with the Transpiler.

Enhancing Transpiler Precision. QUERYHOUSE uses a customized version of the open-source transpiler SQLGlot [34], with modifications aimed at enhancing the precision of semantic equivalence. Ensuring semantic consistency is crucial in our context, as DBMSs often employ different syntax for the same functionality or exhibit different behavior for the same SQL construct. For instance, the sorting behavior of `NULL` values can vary across systems—appearing either first or last. A transpiler handling such subtleties greatly facilitates the automated generation of adapted queries while preserving semantic consistency across target DBMSs.

SQLGlot is inherently designed to support many SQL dialects and to be extensible for new ones. However, during its early integration with QUERYHOUSE, we identified several cases where SQLGlot produced incorrect translations. These issues included:

- **JSON Operator Translation.** The JSON extraction operator (`->`) was mapped to `JSON_EXTRACT` function for SQLite, MySQL, and Oracle, and to `JSON_EXTRACT_PATH` for PostgreSQL. While the translated queries were syntactically correct, their runtime behavior varied across different DBMSs.
- **Data Type Discrepancies.** The `INTEGER` type, for example, occupies 8 bytes in SQLite but only 4 bytes in MySQL, MariaDB, and PostgreSQL, resulting in different overflow behavior. Additionally, SQLGlot translated `INTEGER` to `NUMBER` in Oracle, which represents real numbers rather than integers.
- **Best-Effort Transpilation.** SQLGlot transpiles on a best-effort basis, often leaving unsupported syntax unchanged. This can cause errors due to subtle dialect mismatches. For instance, adding the `RECURSIVE` keyword to Common Table Expressions (CTEs) caused failures in Oracle, where it is not required.

Such subtle discrepancies might be acceptable in SQLGlot’s general use cases, but in our context, they are unacceptable as they can lead to false positives—discrepancies from query mismatches rather than actual DBMS bugs. We enhanced SQLGlot to achieve a satisfactory level of semantic precision by manually verifying translations against official DBMS documentation and refining translation rules for known issues. Unsupported cases without general solutions were excluded. We also extended SQLGlot for previously unhandled syntax, including the `FILTER` clause in window functions and the `STORED` and `VIRTUAL` column types, thereby broadening our testing capabilities. (See §6 and Appendix A.1.)

Extending IR-Based Mutation. Our Hybrid Mutator extends Squirrel’s IR-based mutation mechanism as the baseline mutator. Squirrel’s mutator operates

in two phases: *syntax-preserving mutation*, which modifies the query structure, and *semantics-guided instantiation*, which populates valid values. We incorporate several enhancements into both phases, such as JSON support and improved FOREIGN KEY validity. This enables QUERYHOUSE to generate more syntactically and semantically valid queries, thereby reducing runtime failures.

3.3 LLM-Based Mutation

QUERYHOUSE incorporates a new LLM-based mutation method alongside IR-based mutation within the Hybrid Mutator to improve the quality of SQL generation. This section presents the motivation for adopting an LLM, the metric used to evaluate mutation effectiveness, and its integration method.

Rationale for Using LLMs. The primary motivation for introducing LLMs is to overcome limitations of traditional mutation techniques. Squirrel’s mutator, which QUERYHOUSE uses as a baseline, employs rule-based mutations at the IR level. While IR-based mutation has proven effective for DBMS fuzzing, it applies exactly one mutation per IR. The mutation is based on the IR set merely derived from previously seen input queries, as shown in Fig. 3, without global semantics awareness. This can make it difficult to escape local mutation plateaus and explore semantically diverse query structures, often producing numerous variants with only minor structural differences. We observed that such queries rapidly accumulated in the input queue, making seed selection disproportionately influential and reducing exploration efficiency.

To address these issues, we introduced LLM-based mutations to generate high-quality, diverse inputs that actively enhance exploration. The use of LLMs draws inspiration from recent advances in related fields, such as Text-to-SQL tasks [38,19] and structured data generation for fuzzing [39,25]. Trained extensively on SQL, LLMs can produce well-structured and syntactically valid queries, including novel and structurally diverse forms. By incorporating LLM-based mutations intermittently, QUERYHOUSE aims to achieve broader test coverage through more varied and semantically meaningful query transformations.

Keyword Coverage. Since LLM invocations are both costly and time-consuming, we use them selectively—only when IR-based mutations become ineffective. But how do we determine this point? We employ a metric called *keyword coverage*, originally introduced in SemCont [32] for DBMS conformance testing. Keyword coverage measures the presence of specific SQL keywords in the set of generated queries. While prior work [58,30,26] employed code coverage in single-DBMS settings, instrumenting closed-source, proprietary DBMSs such as Oracle Database is often challenging and inconsistent code coverage reports across different DBMSs can lead to misleading results. In contrast, keyword coverage is well suited for cross-DBMS testing, as it evaluates the input queries themselves rather than depending on DBMS-specific execution behavior.

Monitoring and Integration. The Hybrid Mutator monitors keyword coverage and synchronously triggers an LLM-based mutation when coverage stagnates, introducing only minimal latency (see §5.3). Coverage is computed as the

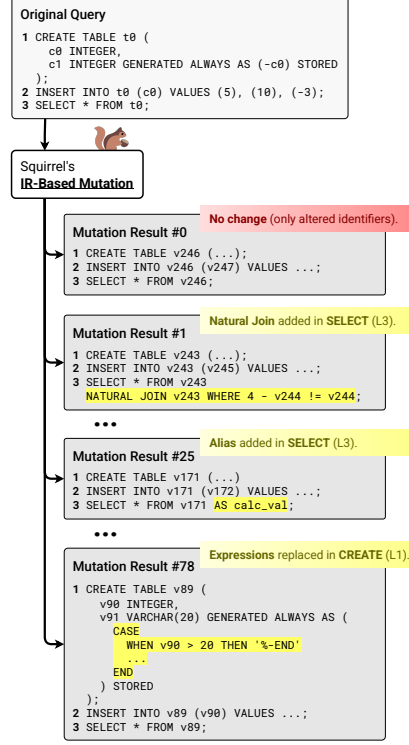


Fig. 3: **Example of Squirrel’s IR-based mutation results.** Structural changes are highlighted by gradient-colored labels. Our review of 78 variants revealed no new statements. Mutation Result #0, without structural changes, likely stemmed from selecting a structurally insignificant IR element.

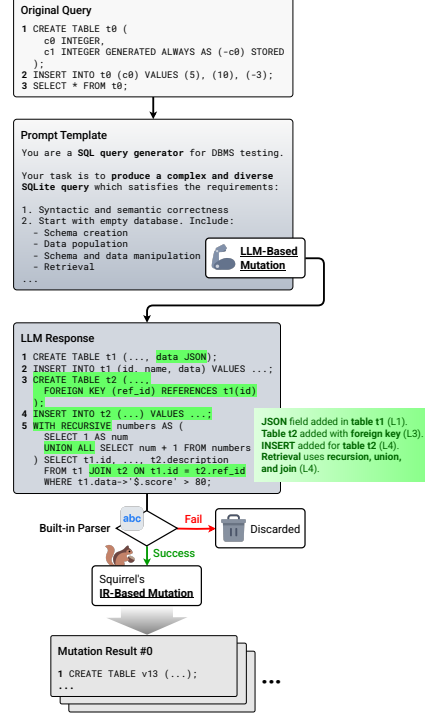


Fig. 4: **Example of LLM-based mutation results.** Structural changes are highlighted. The LLM created a more complex schema with a new JSON field in `t1` and an additional dependent table. The query was diversified with advanced constructs like `RECURSIVE`, `UNION`, and `JOIN` across tables.

number of newly introduced keywords divided by the total size of the keyword set. When this value fails to increase for a predefined number of consecutive mutator calls, QUERYHOUSE identifies stagnation and invokes the LLM using a prompt template that incorporates mutation rules and guidelines. Fig. 4 presents a running example of LLM-based mutation and the resulting operations on the same input base query in Fig. 3. This example demonstrates that the LLM-generated query exhibits substantially greater diversity.

However, one question remains: what if the LLM produces invalid queries? For example, the LLM might introduce features that we have intentionally ex-

cluded for compatibility reasons (see §3.2), despite careful prompt engineering. To address this, we route the LLM-generated queries through our extended IR-based mutator, where invalid queries are filtered out by a built-in parser. By reusing this parser component as an additional validation layer after the constraints provided earlier in the prompt, QUERYHOUSE ensures that LLM-generated queries conform to our constraints and remain compatible with the rest of the pipeline.

3.4 Differential Testing

This section provides a summary of how QUERYHOUSE performs differential testing across target DBMSs. The process involves executing adapted queries on each DBMS and analyzing the results to identify potential bugs.

Query Execution. After the Transpiler stage, we obtain a set of adapted queries ready for execution on each DBMS. The Batch Executor then runs each query on its designated target DBMS and restores the instance to a clean state. This step is essential for the Random Database Approach [20] used in DBMS testing, in which tables and records are created and accessed within a single query. Without resetting, duplicate identifiers in subsequent executions could lead to failures or unintended side effects. While frequent resets may appear to be a performance bottleneck, we mitigate this through *batch execution*, which reduces the number of required resets. Detailed mechanism is provided in §4.

Result Analyzer. We compare results retrieved from target DBMSs as indicators of potential bugs or inconsistencies. However, direct matching may produce false positives, as different results can occur even under correct executions [30]. The SQL standard permits DBMS-specific implementation choices, which can lead to divergent outputs, particularly in data formatting and row ordering. For instance, Boolean values appear as TRUE/FALSE in Oracle Database, t/f in PostgreSQL, and 1/0 in others. To address these variations, we introduced a *normalizer* within the Result Analyzer to enable more permissive and meaningful comparisons. The normalizer reformats DBMS outputs into a unified representation by standardizing formatting for various data types and sorting lexicographically. This step allows QUERYHOUSE to further suppress false positives and focus on truly promising discrepancies likely to be real bugs or inconsistencies.

4 IMPLEMENTATION

We implement QUERYHOUSE, a prototype for cross-DBMS differential testing, built on Squirrel and SQLGlot with customizations to meet our requirements.

Prompt Engineering. We used gpt-4o-2024-08-06 with a prompt template that enforces constraints for testing. This includes SQLite syntax compliance, available data types, semantic correctness, structuring queries into schema creation, data population, manipulation, and retrieval phases, and prohibited constructs and patterns for compatibility using few-shot examples. The full version of our prompt template is available in our repository [2].

Batch Execution. We implemented batch execution to reduce DBMS reset overhead. While Squirrel resets the database after each query, we find this costly for some systems; in Oracle Database, resets—tied to user creation and deletion—incur an average cost of 800 ms per query in our environment.⁸ We reset after a fixed number of queries (100 in our experiments). To prevent collisions, table and column names follow numeric sequences within each batch, preserving self-contained testcases while reducing execution cost.

5 EVALUATION

We evaluate QUERYHOUSE to answer the following research questions.

- **RQ1:** Can QUERYHOUSE detect logic bugs and inconsistencies in real-world DBMSs? (§5.1)
- **RQ2:** How does QUERYHOUSE perform compared to existing works? (§5.2)
- **RQ3:** How do QUERYHOUSE’s transpiler and LLM-based mutation assist differential testing on DBMSs? (§5.3)

Experimental Setup. We evaluated QUERYHOUSE on five DBMSs: Oracle, MySQL, PostgreSQL, SQLite, and MariaDB, all ranked among the top 10 most popular relational DBMSs in both 2024 and 2025 by DB-Engines [51]. Notably, Oracle is a proprietary DBMS that has consistently held the highest rank [13] but has rarely been tested in prior work. The other DBMSs are extensively tested open-source systems [43,23,44,52,27,31]. For bug detection (§5.1), the latest versions available at that time (late 2024) were tested on three laptop PCs owned by the authors: Oracle Database 23ai (23.5.0.24.07), MySQL (8.4~9.1.0), PostgreSQL (REL_17_0), SQLite (3.47.0~3.48.0), and MariaDB (11.6.2). RQ3 experiments (§5.3) ran on two Ubuntu 22.04 servers with two Intel(R) Xeon(R) Gold 6248R CPUs (96 cores) and 256 GB RAM.

5.1 Logic Bug Detection

We conducted a three-week testing campaign with QUERYHOUSE to identify real-world logic bugs in DBMSs. In total, QUERYHOUSE discovered 11 confirmed logic bugs, as detailed in Table 1 (due to space limits, inconsistencies remain in GitHub repository [2]). All identified bugs were reported to their respective DBMS vendors with detailed reports (see §6). Five bugs have been officially fixed. For Bug 8, the vendor updated its official documentation to reflect the operational behavior. The remaining five bugs persist in the latest versions of their respective DBMSs at the time of writing (early 2026), with no corresponding documentation updates. Below, we present two case studies to provide further details and demonstrate the necessity of our differential testing approach for detecting deep, foundational logic bugs.

⁸ The operational timing mentioned is an incidental engineering observation in our specific test environment and does not make any comparative benchmark claims [8].

Table 1: **Detected logic bugs.** QUERYHOUSE detected 11 logic bugs across five real-world DBMSs; all were verified by the respective vendors, and 6 have been fixed.

ID	DBMS	Description	Status
1	SQLite	FILTER not working in window function	Fixed
2	SQLite	REAL constant doubles	Fixed
3	SQLite	ROUND function overflow	Fixed
4	SQLite	SUBSTR function overflow	Fixed
5	MySQL	BETWEEN operator precedence mismatch	Verified
6	MySQL	IN operator precedence mismatch	Verified
7	MySQL	MAX as window aggregate returns NULL	Fixed
8	MariaDB	BETWEEN operator precedence mismatch	Fixed [†]
9	Oracle	LAG/LEAD function overflow	Verified
10	Oracle	NATURAL JOIN on same table results error	Verified
11	Oracle	ROUND function overflow	Verified

[†] MariaDB corrected the documentation to reflect the observed precedence order.

1 SELECT 18446744073709551488;
Expected
Result
sqlite> 1.8446744073709551488e+19

(a) SQLite logic bug incorrectly doubles REAL constant value.

1 CREATE TABLE t0 (c0 INT);
2 INSERT INTO t0 VALUES (1), (2);
3 SELECT v1, LAG(v1, 18446744073709551614, -1)
OVER (ORDER BY v1) FROM v0;
Expected
Result
SQL> V1 LAG(V1,...)OVER...
0 -1
1 -1
2 -1
3 -1
4 -1

(b) Oracle logic bug in LAG/LEAD window aggregate functions.

Fig. 5: Queries exposing bugs in SQLite and Oracle, identified by QUERYHOUSE.

1. **SQLite Constant Doubles.** We identified a logic bug in SQLite that parses a REAL constant as a double when the value falls within a specific range (Bug 2). This bug resulted from an internal decision to remove the “long double” data type for compatibility, and QUERYHOUSE detected it within two months of the bug-introducing commit. Despite the simplicity of its PoC, as shown in Fig. 5a, we expect that prior testing methods would have difficulty detecting this bug. Because the bug originates in core DBMS logic, sympathetic failures would have been unavoidable regardless of how the query was transformed.
2. **Oracle LAG/LEAD Overflow.** LAG/LEAD are window functions used to access rows at a specified offset backward or forward within a window frame. We discovered an overflow bug in Oracle in which, given a large boundary value (e.g., $2^{64} - 1$), the operations fail as shown in Fig. 5b. Because inducing invariants for window functions is difficult due to their complex internal mechanisms, designing metamorphic oracles for them is highly challenging. Dif-

Table 2: Detection of bugs by existing tools

Bug	SQLite			MySQL			Maria		Oracle		
	1	2	3	4	5	6	7	8	9	10	11
NoREC	✗	✗	✗	✗	✗*	✗*	✗*	✗	✗*	✗*	✗*
TLP	FP	✗	✗	✗	✗	✗	✗	✗*	FP*	✗*	✗*
DQP	✗*	✗*	✗*	✗*	✗	✗	✗	✗	✗*	✗*	✗*

TP = true positive; ✗ = false negative; FP = false positive.

* The oracle does not natively support the corresponding feature or DBMS; these cells force-run the extended oracle with appropriate dialect adjustments.

ferential testing with QUERYHOUSE serves as a practical surrogate. Notably, to our knowledge, prior work has identified only memory bugs in window functions [42], but no logical bugs.

False Positives. Since DBMSs pursue considerable flexibility in their design choices, such semantic differences are often documented as caveats and can cause false positives in differential testing. However, some are lesser-known and require care; we believe QUERYHOUSE can also help detect such disagreements. For example, one false positive involved the TEMPORARY table. We observed that in Oracle, DDL queries operate on TEMPORARY tables but silently discard data without errors or warnings, in contrast to other DBMSs. Manual triage revealed this behavior was intentional—Oracle implicitly commits before DDL—after surveying various materials about its session- and transaction-specific behavior. Although documented, this behavior may be unfamiliar to users of other DBMSs and can lead to unintended consequences if overlooked.

5.2 Comparison with Existing Tools

To demonstrate the effectiveness and novelty of QUERYHOUSE, we compare its bug detection capability against existing approaches. Specifically, we evaluate whether prior metamorphic oracles, NoREC, TLP, and DQP, can detect the bugs identified by QUERYHOUSE. We supply the exact SQL queries that trigger each bug to these oracles, assuming a favorable setting in which they have already discovered the problematic queries, and execute them on real DBMSs. Because these approaches are generation-based and do not provide interfaces for testing a given query, we reimplement their transformations based on the code and published descriptions. For a meaningful comparison, we extend the prior approaches to support additional features and DBMS dialects beyond original implementations whenever doing so does not require fundamental changes to the transformation logic, thereby strengthening the baselines.

As summarized in Table 2, the experimental results show that none of the bugs are correctly detected by these oracles: either no alarms are raised, or the buggy and patched DBMSs cannot be distinguished due to false alarms. In most

Table 3: **Validity Results.** All scores are averages of five experiments measured at the end of a 24-hour run. Δ Validity denotes the validity of QUERYHOUSE relative to QUERYHOUSE_{-T}. Since the base query uses SQLite syntax, query validity remains unchanged with and without the transpiler.

	DBMS					valid@k	
	SQLite	Oracle	MySQL	PSQL	Maria	valid@5	valid@2
QUERYHOUSE	0.4883	0.2168	0.2495	0.2030	0.2747	0.1633	0.2795
QUERYHOUSE _{-T}	0.4883	0.1740	0.1096	0.1660	0.0806	0.0457	0.2395
Δ Validity	–	24.56%	127.64%	22.29%	241.05%	257.03%	16.72%

cases (**X**), both the original and transformed queries return identical incorrect results, preventing MT oracles from detecting anomalies. In two cases (FP), the oracles raise false alarms: (Bug 1, TLP) reports discrepancies in both the buggy and fixed DBMSs. Since no fixed DBMS is available for (Bug 9, TLP), we manually confirm that it exhibits the same pattern as (Bug 1, TLP). These results indicate that QUERYHOUSE detects a fundamentally distinct class of bugs from existing methods, underscoring the novelty and necessity of our approach.

5.3 Transpiler and LLM-based Mutation Effects

We conducted an ablation study to evaluate the roles of QUERYHOUSE’s key components: the transpiler and LLM-based mutation. To assess the transpiler, we measured query validity—the proportion of testcases executed without syntax or semantic errors—by comparing QUERYHOUSE with QUERYHOUSE_{-T}, a variant without the transpiler. We report valid@k, the proportion of testcases valid across at least k DBMSs, focusing on valid@5 and valid@2 to capture queries supported by all or at least two DBMSs. For the LLM-based mutation, we compared keyword coverage between QUERYHOUSE and QUERYHOUSE_{-L}, which disables LLM-based mutation, to evaluate its impact on exploration diversity and input quality. All experiments ran for 24 hours and were repeated five times.

Transpiler Results. Table 3 reports averaged per-DBMS validity and valid@5 / valid@2 after 24 hours. We observed that with transpilation, QUERYHOUSE consistently achieves higher validity than QUERYHOUSE_{-T} in each DBMS. For validity across multiple DBMSs, valid@5 of QUERYHOUSE is 0.1633, representing a 257.03% improvement over QUERYHOUSE_{-T}, a substantial gain in cross-DBMS testing where validity is inherently constrained by shared features. This underscores the importance of high-precision transpilation in differential testing. QUERYHOUSE_{-T} shows a higher valid@2 (0.2395) than valid@5, reflecting partial syntax overlap across DBMSs. Still, QUERYHOUSE achieves a higher valid@2 (0.2795), demonstrating its effectiveness even when such overlap is insufficient.

LLM-based Mutation Results. Fig. 6 shows keyword coverage over time. QUERYHOUSE achieves slightly higher coverage than QUERYHOUSE_{-L} (84.03%

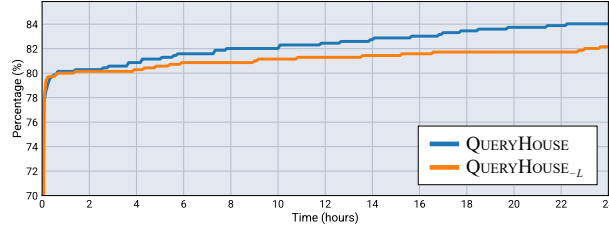


Fig. 6: Keyword coverage measured over time.

vs. 82.16%) and increases more rapidly, with a gain of 3.89 percent points (pp) after one hour compared to 2.16 pp for `QUERYHOUSE-L`. Most unreached keywords were either excluded due to inconsistent DBMS support (e.g., `GROUPS`) or were non-standard, DBMS-specific (e.g., `TOP`, `OTHERS`). We expect further improvements in transpiler support and LLM ability could increase coverage.

LLM cost. To assess the cost of LLM integration, we measured API cost and response time during a 24-hour campaign. On average, `QUERYHOUSE` consumed 159K input tokens and 72K output tokens, resulting in a total API cost of \$1.12. The cumulative response time accounted for only 1.68% of the total runtime (24.2 minutes). These results indicate that LLM-based mutation incurs minimal cost and latency overhead, supporting its viability for real-world deployment.

6 DISCUSSION

Responsible Disclosure. We reported all discovered bugs with detailed reports to the respective DBMS vendors. In the case of Oracle Database, which restricts its report channel to users with an active support contract, we submitted our report via the Oracle Developer Community following forum members’ advice and received a response from moderators. We also opened our full list of confirmed bug reports on our GitHub repository.

Supporting New Targets. `QUERYHOUSE` significantly reduces the implementation overhead required to support new target DBMSs, in contrast to prior works [43,30,23,52] that rely heavily on target-specific engineering [31]. For example, supporting a new DBMS in Squirrel requires (1) customizing a general parser to accommodate the unique features of the DBMS and (2) writing semantic relation rules according to the target grammar. The Squirrel authors estimated that these tasks would take up to two days [58]. In contrast, `QUERYHOUSE` dramatically reduces this effort through its differential testing approach. Leveraging SQLGlot, which already supports 27 SQL dialects, `QUERYHOUSE` typically requires only a small amount of code to extend support for a new DBMS. For example, after we built `QUERYHOUSE` targeting four DBMSs—Oracle, MySQL, PostgreSQL, and SQLite—we decided to extend support to MariaDB, and the additional workload was minimal (<60 LoC), as shown in Table 4. Although addressing SQLGlot’s precision issues may require minor additional effort (see §3.2),

Table 4: **Line of Code (LoC) to support MariaDB.** Changes unrelated to QUERYHOUSE features, such as DBMS client or build files, were excluded. No modifications were made to SQLGlue due to the shared use of MySQL syntax.

Component	Source Files	Added LoC
Batch Executor	<code>internal/client/client.cc</code>	9
	<code>protocol.{cc,h}</code>	12
Transpiler	<code>transpiler.cc</code>	8
DBMS Configuration	<code>config/template_mariadb.yml</code>	22

such enhancements are relatively simple compared to the engineering demands of Squirrel. Based on our experience, these adjustments were completed within an hour. This highlights the scalable design of QUERYHOUSE, which enables efficient and low-effort support for new target DBMSs.

7 RELATED WORK

Testing for Logic Bugs. Many studies have explored methods for testing logic bugs in DBMSs. A common approach is the development of bug oracles. Khalek et al. [4] and Marcozzi et al. [35] employed SAT solvers to determine the ground truth for queries. PQS [46] synthesized queries that select a random row as a pivot. SemCont [32] formally defined SQL semantics to construct a conformance oracle. EET [27] generated equivalent queries at the expression level. Metamorphic testing [43,30] has also been widely studied, verifying query correctness by comparing the results of semantically equivalent or transformed queries using designated bug oracles [46,44,45,52,7,23,57]. Recently, QTRAN [31] utilized an LLM with retrieval-augmented generation (RAG) to deploy bug oracles for metamorphic testing in multiple DBMSs. Compared to these approaches, QUERYHOUSE can detect logic bugs even when a DBMS feature is fundamentally broken or exhibits complex behavior that makes it difficult to design metamorphic transformation rules while preserving semantic relations.

Additionally, prior work has applied differential testing [36,15,11] to assess query correctness across different DBMSs, which is closest to our approach. RAGS [50] stochastically generates basic SQL statements such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, and executes them on multiple DBMSs. The `sqllogictest` [24] feature of SQLite compares the results of a fixed query set against a reference DBMS. SparkFuzz [21], which aimed to identify concurrency bugs in the Apache Spark engine, used PostgreSQL as a reference DBMS and compared other DBMSs’ query results. DT² [11] utilized differential testing on three MySQL-compliant DBMSs to find bugs in transactional executions, a relatively well-standardized feature. QUERYHOUSE is inspired by and extends this lineage. Unlike previous works, QUERYHOUSE leverages a transpiler to address syntactic differences across DBMSs during query generation in a scalable manner, thereby enabling differential testing for a broader spectrum of SQL features.

Memory and Performance Bugs. Extensive research has also been conducted on detecting other types of bugs, including crashes, memory corruption, and performance issues. sqlsmith [47] applies generation-based fuzzing by creating queries based on the Abstract Syntax Tree (AST) of a fixed schema. DynSQL [26] improves query validity by dynamically leveraging state information during query generation. Squirrel [58] enhances query validity and fuzzing efficiency through IR-level mutations combined with semantics-aware instantiation. For performance bug detection, BmPad [41] identifies queries that exceed an estimated time threshold. Similarly, AMOEBA [33] and Apollo [28] execute automatically generated queries and detect pairs that exhibit significant differences in execution time. We expect that the methods of QUERYHOUSE—use of a transpiler and LLM-based mutation—can independently aid these works by expanding them to a multi-DBMS context and facilitating broader exploration.

8 CONCLUSION

Despite recent efforts to identify logic bugs in DBMSs, the limitations of prior methods and the SQL discrepancy problem have necessitated per-DBMS and per-bug type engineering, leaving blind spots for deep bugs rooted in implementation flaws. We present QUERYHOUSE, a novel automated framework that enables broad differential testing across SQL features using LLM-based mutation and query transpilation. We evaluated QUERYHOUSE on five DBMSs—Oracle, MySQL, PostgreSQL, SQLite, and MariaDB—and found 11 verified logic bugs and 7 inconsistencies. Our evaluation confirmed that transpilation, LLM-based mutation, and batch execution each contributed to improved query validity, coverage, and speed. We hope QUERYHOUSE will foster further discussion and new directions in the DBMS testing community.

Acknowledgments. We thank the anonymous reviewers for providing valuable feedback. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korean government(MSIT) (RS-2026-25474019). This research was also supported by the “Best of the Best” (BoB) program, conducted by the Korea Information Technology Research Institute (KITRI). We also thank Changheon Lee for infrastructure support.

Data Availability. All source code and experimental artifacts are publicly available on GitHub [2,3] to support replication, future research, and practical use.

A Appendix

A.1 Transpiler Enhancement

Table 5 summarizes the modifications made to SQLGlot for cross-DBMS testing in QUERYHOUSE. Upon reviewing 13 feature categories, we revised 14 transpilation rules, including 10 fixes and 4 additions. Most required minimal engineering effort based on keyword substitution, such as support for data types and new keywords. Three features were excluded because of inconsistent support across multiple DBMSs.

Table 5: Transpilation rules fixed or added per feature category

Category	Rules fixed/added
Data types	6 fixed (TEXT, VARCHAR(n), CHAR(n), INT, FLOAT, DOUBLE)
Window function	1 fixed (FILTER)
Generated column	3 added (GENERATED ALWAYS AS, VIRTUAL, STORED)
Foreign key	1 fixed (ON DELETE NO ACTION)
JSON	1 fixed (JSON extractor)
CTE	1 fixed (RECURSIVE)
View	None
Temporary table	None
Sorting	None
Join	1 added (USING clause)
Partitioning	Excluded
Returning	Excluded
FTS	Excluded

References

1. ISO/IEC 9075: Information technology — Database languages — SQL (2016), <https://www.iso.org/standard/63555.html>, available in multiple parts, e.g., SQL/Foundation, SQL/CLI, SQL/PSM, etc.
2. QueryHouse. <https://github.com/queryhouse-2037/QueryHouse> (2026)
3. QueryHouse-SQLGlot. <https://github.com/queryhouse-2037/QueryHouse-sqlglot> (2026)
4. Abdul Khalek, S., Khurshid, S.: Automated sql query generation for systematic testing of database engines. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering. pp. 329–332 (2010)
5. Adya, A.: Weak consistency: a generalized theory and optimistic implementations for distributed transactions (1999)
6. Agrawal, R., Somani, A., Xu, Y.: Storage and querying of e-commerce data. In: VLDB. vol. 1, pp. 149–158 (2001)
7. Ba, J., Rigger, M.: Keep it simple: Testing databases via differential query plans. Proceeding of ACM Management of Data (SIGMOD’24) (Jun 2024)
8. Belmas, G.I., Larson, B.N.: Clicking away your speech rights: The enforceability of gagwrap licenses. Communication Law and Policy **12**(1), 37–89 (2007)
9. Ceruti, M.G.: Data management challenges and development for military information systems. IEEE Transactions on Knowledge and Data Engineering **15**(5), 1059–1068 (2003)
10. Coronel, C., Morris, S.: Database systems: design, implementation and management. Cengage learning (2019)
11. Cui, Z., Dou, W., Dai, Q., Song, J., Wang, W., Wei, J., Ye, D.: Differentially testing database transactions for fun and profit. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. pp. 1–12 (2022)
12. Date, C.: A critique of the sql database language. ACM Sigmod Record **14**(3), 8–54 (1984)

13. Edith: Oracle ranks as the world's most popular database management system with a 1,266 ranking score. <https://stockapps.com/blog/oracle-ranks-as-the-worlds-most-popular-database-management-system-with-a-1266-ranking-score/> (2022)
14. Eisenberg, A., Melton, J.: Sql standardization: the next steps. *ACM SIGMOD Record* **29**(1), 63–67 (2000)
15. Evans, R.B., Savoia, A.: Differential testing: a new approach to change detection. In: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers. pp. 549–552 (2007)
16. Florescu, D., Levy, A., Mendelzon, A.: Database techniques for the world-wide web: A survey. *ACM Sigmod Record* **27**(3), 59–74 (1998)
17. Forum, S.U.: Filter clause is ignored under certain conditions during window functions. <https://sqlite.org/forum/forumpost/e9126d554a> (2024)
18. Fu, J., Liang, J., Wu, Z., Zhao, Y., Li, S., Jiang, Y.: Understanding and detecting sql function bugs: Using simple boundary arguments to trigger hundreds of dbms bugs. In: Proceedings of the Twentieth European Conference on Computer Systems. pp. 1061–1076 (2025)
19. Gao, D., Wang, H., Li, Y., Sun, X., Qian, Y., Ding, B., Zhou, J.: Text-to-sql empowered by large language models: A benchmark evaluation. *CoRR* **abs/2308.15363** (2023)
20. Gao, X., Liu, Z., Cui, J., Li, H., Zhang, H., Wei, K., Zhao, K.: A comprehensive survey on database management system fuzzing: Techniques, taxonomy and experimental comparison. *arXiv preprint arXiv:2311.06728* (2023)
21. Ghit, B., Poggi, N., Rosen, J., Xin, R., Boncz, P.: Sparkfuzz: Searching correctness regressions in modern query engines. In: Proceedings of the workshop on Testing Database Systems. pp. 1–6 (2020)
22. Haigh, T.: How charles bachman invented the dbms, a foundation of our digital world. *Communications of the ACM* **59**(7), 25–30 (2016)
23. Hao, Z., Huang, Q., Wang, C., Wang, J., Zhang, Y., Wu, R., Zhang, C.: Pinolo: Detecting logical bugs in database management systems with approximate query synthesis. In: 2023 USENIX Annual Technical Conference (USENIX ATC 23). pp. 345–358 (2023)
24. Hipp, D.R.: sqllogictest Documentation. <https://www.sqlite.org/sqllogictest/doc/trunk/about.wiki> (2008)
25. Hu, J., Zhang, Q., Yin, H.: Augmenting greybox fuzzing with generative ai. *arXiv preprint arXiv:2306.06782* (2023)
26. Jiang, Z.M., Bai, J.J., Su, Z.: DynSQL: Stateful fuzzing for database management systems with complex and valid SQL query generation. In: Proceedings of the 32nd USENIX Security Symposium (Security). Anaheim, CA (Aug 2023)
27. Jiang, Z.M., Su, Z.: Detecting logic bugs in database engines via equivalent expression transformation. In: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24). pp. 821–835 (2024)
28. Jung, J., Hu, H., Arulraj, J., Kim, T., Kang, W.: Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment* **13**(1), 57–70 (2019)
29. Liang, J., Wu, Z., Fu, J., Bai, Y., Zhang, Q., Jiang, Y.: {WingFuzz}: Implementing continuous fuzzing for {DBMSs}. In: 2024 USENIX Annual Technical Conference (USENIX ATC 24). pp. 479–492 (2024)
30. Liang, Y., Liu, S., Hu, H.: Detecting logical bugs of {DBMS} with coverage-based guidance. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 4309–4326 (2022)

31. Lin, L., Zhu, Q., Chen, H., Wang, Z., Wu, R., Xie, X.: Qtran: Extending metamorphic-oracle based logical bug detection techniques for multiple-dbms dialect support. *Proceedings of the ACM on Software Engineering* **2**(ISSTA), 731–752 (2025)
32. Liu, S., Tian, C., Sun, J., Wang, R., Lu, W., Zhao, Y., Xue, Y., Wang, J., Du, X.: Semantic conformance testing of relational dbms. *Proceedings of the VLDB Endowment* **18**(3), 850–862 (2024)
33. Liu, X., Zhou, Q., Arulraj, J., Orso, A.: Automatic detection of performance bugs in database systems using equivalent queries. In: *Proceedings of the 44th International Conference on Software Engineering*. pp. 225–236 (2022)
34. Mao, T.: Sqlglot. <https://github.com/tobymao/sqlglot> (2021)
35. Marcozzi, M., Vanhoof, W., Hainaut, J.L.: Test input generation for database programs using relational constraints. In: *Proceedings of the Fifth International Workshop on Testing Database Systems*. pp. 1–6 (2012)
36. McKeeman, W.M.: Differential testing for software. *Digital Technical Journal* **10**(1), 100–107 (1998)
37. McWherter, D.T., Schroeder, B., Ailamaki, A., Harchol-Balter, M.: Priority mechanisms for oltp and transactional web applications. In: *Proceedings. 20th International Conference on Data Engineering*. pp. 535–546. IEEE (2004)
38. Nan, L., Zhao, Y., Zou, W., Ri, N., Tae, J., Zhang, E., Cohan, A., Radev, D.: Enhancing text-to-sql capabilities of large language models: A study on prompt design strategies. In: *Findings of the Association for Computational Linguistics: EMNLP 2023*. pp. 14935–14956 (2023)
39. Oliinyk, Y., Scott, M., Tsang, R., Fang, C., Homayoun, H., et al.: Fuzzing {BusyBox}: Leveraging {LLM} and crash reuse for embedded bug unearthing. In: *33rd USENIX Security Symposium (USENIX Security 24)*. pp. 883–900 (2024)
40. Pentland, A.S.: The data-driven society. *Scientific American* **309**(4), 78–83 (2013)
41. Rehmann, K.T., Seo, C., Hwang, D., Truong, B.T., Boehm, A., Lee, D.H.: Performance monitoring in sap hana’s continuous integration process. *ACM SIGMETRICS Performance Evaluation Review* **43**(4), 43–52 (2016)
42. Rigger, M.: Bugs found in Database Management Systems. <https://www.manuelrigger.at/dbms-bugs/>
43. Rigger, M.: SQLancer. <https://github.com/sqlancer/sqlancer> (2021)
44. Rigger, M., Su, Z.: Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In: *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2020* (2020). <https://doi.org/10.1145/3368089.3409710>
45. Rigger, M., Su, Z.: Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* **4**(OOPSLA) (2020). <https://doi.org/10.1145/3428279>
46. Rigger, M., Su, Z.: Testing database engines via pivoted query synthesis. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Banff, Alberta (Nov 2020), <https://www.usenix.org/conference/osdi20/presentation/rigger>
47. Seltenreich, A., Tang, B., Mullender, S.: SQLsmith: a random SQL query generator. <https://github.com/anse1/sqlsmith>
48. Silberschatz, A., Korth, H.F., Sudarshan, S.: *Database system concepts* (2011)
49. Silberschatz, A., Stonebraker, M., Ullman, J.: Database systems: Achievements and opportunities. *Communications of the ACM* **34**(10), 110–120 (1991)
50. Slutz, D.R.: Massive stochastic testing of sql. In: *VLDB*. vol. 98, pp. 618–622. Citeseer (1998)

51. Software, R.: Db-engines ranking. <https://db-engines.com/en/ranking> (2025)
52. Song, J., Dou, W., Cui, Z., Dai, Q., Wang, W., Wei, J., Zhong, H., Huang, T.: Testing database systems via differential query execution. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). pp. 2072–2084. IEEE (2023)
53. Sotiropoulos, T., Chaliasos, S., Atlidakis, V., Mitropoulos, D., Spinellis, D.: Data-oriented differential testing of object-relational mapping systems. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 1535–1547. IEEE (2021)
54. SQLite: Appropriate uses for sqlite. <https://www.sqlite.org/whentouse.html> (2015)
55. SQLite: Sql features that sqlite does not implement. <https://www.sqlite.org/omitted.html> (2022)
56. System, M.B.: Max() as a window function returns null for the first row even with non-empty window frame under specific conditions. <https://bugs.mysql.com/bug.php?id=117072> (2024)
57. Zhang, C., Rigger, M.: Constant optimization driven database system testing. *Proceedings of the ACM on Management of Data* **3**(1), 1–24 (2025)
58. Zhong, R., Chen, Y., Hu, H., Zhang, H., Lee, W., Wu, D.: Squirrel: Testing database management systems with language validity and coverage feedback. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. pp. 955–970 (2020)
59. Zhou, W., Gao, Y., Zhou, X., Li, G.: Cracking sql barriers: An llm-based dialect translation system. *Proceedings of the ACM on Management of Data* **3**(3), 1–26 (2025)