ACME: Automated Clause Mapping Engine for Testing Emerging Database Systems

YUANCHENG JIANG, National University of Singapore, Singapore
JIANING WANG, Shandong University, China
CHUQI ZHANG, National University of Singapore, Singapore
ROLAND H. C. YAP, National University of Singapore, Singapore
ZHENKAI LIANG, National University of Singapore, Singapore
MANUEL RIGGER, National University of Singapore, Singapore

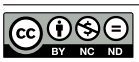
A growing number of emerging database management systems, such as time-series and streaming database systems, have been developed to support specialized workloads with enhanced performance and functionality. However, these systems are often less mature than traditional relational database systems, making them more prone to logic bugs and internal errors affecting correctness and reliability. To address this, we propose an enhanced differential testing framework designed for emerging SQL-like database systems. Our key insight is that many of these systems are conceptually extensions of relational database systems, allowing us to uncover bugs by comparing query results with those from more robust and mature relational database systems. To bridge the differences in syntax and semantics between emerging and relational database systems, we leverage Large Language Models (LLMs) to make differential testing more effective by discovering clause mappings that translate system-specific features in emerging database systems into equivalent SQL expressions. Our approach proceeds in three steps: (i) analyzing the syntax and semantics of queries with runtime errors to reason on clause mappings using LLMs; (ii) validating the generated clause mappings by executing test queries and re-prompting upon validation failures; and (iii) generating semantically equivalent, yet syntactically diverse queries to broaden the coverage of differential testing. We implemented this approach in a tool called `ACME` and applied it to four widely used emerging database systems, uncovering 59 previously unknown bugs, including 17 logic bugs and 42 internal errors. Of these, 52 have been fixed and 5 confirmed by vendors. Our evaluation demonstrates that `ACME` enhances LLM reliability through query validation. Furthermore, the evaluation shows the effectiveness of differential testing even when using local models or a limited online token budget. Our results demonstrate the practicality and effectiveness of `ACME` in improving the robustness and accuracy of emerging database systems through scalable, LLM-assisted differential testing.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: DBMS Testing, Differential Testing, LLM-based Testing

ACM Reference Format:
Yuancheng Jiang, Jianing Wang, Chuqi Zhang, Roland H. C. Yap, Zhenkai Liang, and Manuel Rigger. 2026. ACME: Automated Clause Mapping Engine for Testing Emerging Database Systems. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE039 (July 2026), 23 pages. https://doi.org/10.1145/3797134

Authors' Contact Information: Yuancheng Jiang, yuancheng@comp.nus.edu.sg, National University of Singapore, Singapore; Jianing Wang, jianingwang@mail.sdu.edu.cn, Shandong University, China; Chuqi Zhang, chuqiz@comp.nus.edu.sg, National University of Singapore, Singapore; Roland H. C. Yap, ryap@comp.nus.edu.sg, National University of Singapore, Singapore; Zhenkai Liang, liangzk@comp.nus.edu.sg, National University of Singapore, Singapore; Manuel Rigger, rigger@nus.edu.sg, National University of Singapore, Singapore.



ACM 2994-970X/2026/7-ARTFSE039
https://doi.org/10.1145/3797134

1 Introduction

Recently, the landscape of database management systems (DBMS) has witnessed a significant transformation with the emergence of diverse and specialized database systems (*i.e.,* emerging database systems). These emerging database systems have been developed due to the evolving needs of modern applications and the complexities of handling specific workloads. Unlike traditional ones, emerging database systems are purpose-built to address specific use cases across a range of industries and applications.

Emerging database systems, often developed under market pressures, are typically less mature and thus more susceptible to bugs than established relational database systems. Testing such systems seeks to reveal both internal errors and logic bugs. Internal errors (*e.g.,* crashes, exceptions, or unexpected aborts) are relatively straightforward to detect. In contrast, logic bugs are more challenging: they produce incorrect results without obvious signs of failure, and therefore escape the notice of both developers and users. While manually written tests for a single database system can expose logic bugs, this approach remains an unscalable solution for the generalized problem of logic bug discovery. The central challenge is whether an automated and effective testing strategy can be devised for diverse emerging database systems, each with distinct syntax and semantics.

A scalable and generalized testing strategy hinges on the design of an effective test oracle. In this work, our core insight is that emerging database systems can be viewed as extensions of relational database systems, enabling the discovery of bugs through differential testing against well-established relational systems, which have been extensively validated [5, 24, 30, 41–43, 57] and are known for their robustness. Nevertheless, a practical challenge arises: differential testing would require queries with equivalent semantics. Unlike traditional relational models, emerging database systems introduce distinct syntactic and semantic variations. These include the implementation of entirely new query clauses, as well as "common clauses" that share existing syntax but yield different semantic results.

To address this challenge, we propose *Automated Clause Mapping Engine* (`ACME`), a novel and practical approach that extends the scope of differential testing in emerging database systems by mapping new or altered clauses in the emerging database to mature relational database systems. To automatically create the mappings that lead to an effective differential testing oracle, `ACME` leverages Large Language Models (LLMs) as a knowledge bridge between traditional and emerging database systems to create clause mappings. The approach incorporates failed test query pairs (*i.e.,* the query failing on only one database instance) and their associated error messages. It prompts LLMs to generate clause mappings as solutions by providing them with candidate mappings and corresponding validation queries. `ACME` then incorporates test-query validation via test queries (minimal SQL queries that check the semantics of clauses with the corresponding expected results) and error-driven re-prompting to iteratively resolve invalid clause mappings, thereby improving the correctness of the final output. By using *clause mappings*, `ACME` automatically generates semantic-equivalent SQL queries, giving more code coverage than regular differential testing, expanding the scope of differential testing. Although syntactically different, these queries should produce identical results. Any discrepancy in the results indicates a logic bug in the emerging system, assuming correctness in the mature reference system.

Listing 1 demonstrates how our approach discovers an unknown bug[1] in emerging database systems via clause mappings. The initial query Q checks whether a value is either zero or `null`. While query execution yields different results in QuestDB (an emerging database) and a relational database (PostgreSQL), this difference is not a bug and is expected because the semantics of `null` is also

[1] https://github.com/questdb/questdb/issues/4505, https://github.com/questdb/questdb/issues/4528

```
CREATE TABLE test (c0 INT); -- Initialize Schema
INSERT INTO test VALUES (NULL); -- Initialize Data
Q: SELECT (c0 IN (0, NULL)) FROM test; -- Original Query for PostgreSQL
Q: SELECT (c0 IN (0, NULL)) FROM test; -- Original Query for QuestDB
-- PostgreSQL: [(NULL)] != QuestDB: [(True)] -- (This leads to a false alarm)

-- Clause Mapping in PostgreSQL with COALESCE --
Q1(mapped): SELECT (COALESCE(c0::TEXT,'_NULL_') IN (0::TEXT,'_NULL_')) FROM test;
-- PostgreSQL: [(True)] != QuestDB: [(True)] -- (This prevents the false alarm)

-- Clause Mapping in QuestDB with CASE WHEN --
Q2(mapped): SELECT (CASE WHEN c0 IS NULL THEN NULL ELSE c0 IN (0) END) FROM test;
-- PostgreSQL: [(NULL)] != QuestDB: [(False)] -- (Accepted as a valid bug)
```

Listing 1. Unknown Logic Bug ACME Found in QuestDB through Clause Mappings

different between QuestDB and SQL.[2] Nevertheless, this difference is expected after investigating their distinct treatments of null. Under conventional differential testing, this would either be flagged as a false alarm or avoided by excluding such features (*e.g.,* disallowing null values), which risks missing null-related bugs and further narrowing the testing scope. **ACME** addresses this gap using clause mappings. As shown in Q_1 *mapped*, we translate the null clause into a COALESCE expression in PostgreSQL, using a sentinel to emulate QuestDB's "null as a value" semantics. This enables differential testing to handle queries involving null values while preserving cross-system semantic equivalence, thereby preventing false alarms. Clause mappings can be applied in both directions. If we consider mapping the QuestDB query to PostgreSQL as shown in Q_2 *mapped*, **ACME** maps the in clause to a case...when clause. This allows us first to handle null values and otherwise return the boolean value for non-null expressions. If the difference persists after mapping, we identify a bug-inducing case. In this case, the QuestDB developers promptly acknowledged the issue.

To assess the effectiveness of our technique, we used **ACME** to test four popular emerging database systems: QuestDB [39], TDEngine [48], RisingWave [44], and CrateDB [14]. All selected emerging database systems have been growing in popularity (*e.g.,* having at least thousands of stars on Github) and are actively maintained and updated. We found a total of 59 previously unknown bugs (17 logic bugs and 42 internal errors) where 52 of them have been fixed and 5 confirmed. We evaluated **ACME** on its improvements in bug finding for SQL-like emerging database systems: (i) demonstrating its effectiveness in finding more unknown bugs; and (ii) greater coverage with more unique query plans. We also compared **ACME** against the state-of-the-art test oracle, Ternary Logic Partition (TLP) [42], and its bug-finding tool SQLancer [43]. Our experiments show that **ACME** is more effective in detecting bugs (*e.g.,* it identified 17 logic bugs that SQLancer was unable to find) and explores greater code coverage when testing SQL-like emerging database systems. To assess LLM effectiveness and feasibility, we show that **ACME** 's test-query validation and re-prompting improves the correctness of generated mappings. Moreover, ACME consistently enhances the effectiveness of differential testing even under a lightweight LLM budget, whether using local models or limited online tokens. Our efforts received encouraging acknowledgments from QuestDB developers in their official post blog.[3]

In summary, we make the following contributions:

- We present our key insight that emerging database systems are often extensions of relational ones, enabling extended differential testing via clause mappings across heterogeneous database systems having partially incompatible database query language dialects.

[2]QuestDB regards null as a specific value, while PostgreSQL always returns null when comparing null expression.
[3]Acknowledged at https://questdb.io/blog/fuzz-testing-questdb.

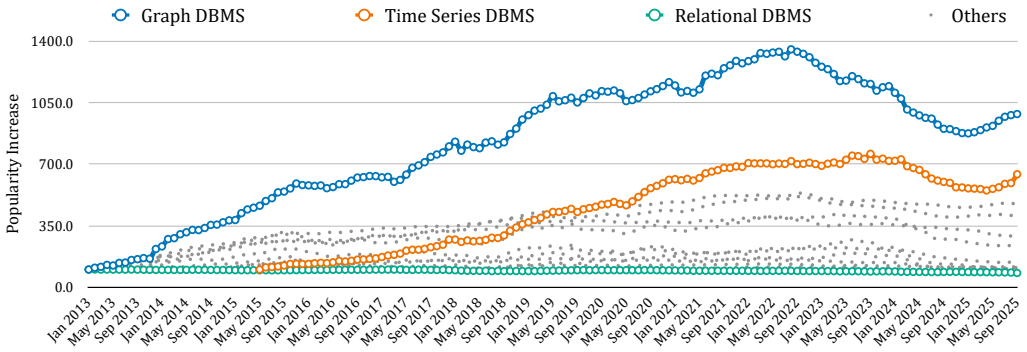


Fig. 1. Popularity Trend of Database Systems Past Decade

- We develop a practical tool called **ACME**,[4] which leverages LLMs as a knowledge bridge across heterogeneous database systems. By producing semantically equivalent queries expressed in divergent syntax, **ACME** enables automated and scalable testing across database systems.
- **ACME** demonstrates its effectiveness by finding 59 previously unknown bugs across four popular systems, with 52 fixes and 5 confirmations to date.

2 Background

This section provides background on emerging database systems to (i) motivate the need for testing them and (ii) present an intuition for our testing approach. We begin by examining the scope of recently introduced systems and studying their adoption trends. We then compare their feature sets, observing that many "new" features can be interpreted as extensions of existing expressions in relational database systems. Then, we survey their query languages and interfaces.

In summary, we investigate the following motivating questions (Q):

- *Q1: What are the types of emerging database systems, and what is their popularity?*
- *Q2: What are new features in emerging database systems?*
- *Q3: What are their query languages?*

To systematically examine emerging database systems, we collect recent statistics from major survey platforms, including *DB-Engines* [16] and the *database of databases* (dbdb) [15] which provide up-to-date information on emerging database systems, including popularity, data models, and query languages. We also refer to their official documentation and open-source platforms (*e.g.,* GitHub) for supplementary statistics and information.

Q1: Types and popularity. Our first question is: *What types of emerging database systems exist, and how popular are they?* We observe a steady rise in the adoption of such systems, largely driven by specialized data models and new features tailored to particular workloads. Additionally, because these systems often feature newly developed and less-vetted codebases, implementing scalable and rigorous testing is essential for maintaining reliability. In this paper, we distinguish between (a) *conventional* database systems and (b) *emerging* database systems, focusing on those providing SQL-like query languages.

Conventional database systems refer to relational database systems, which have evolved over decades and remain the predominant choice in practice due to their reliability and robustness. They implement the relational model (representing data as tables with well-defined schemas and integrity constraints) and provide a mature SQL ecosystem. Representative systems include MySQL [36], SQLite [47], and PostgreSQL [38]. Many works have investigated testing methodologies

[4] **ACME** is available at https://github.com/YuanchengJiang/ACME.

```
SELECT ts,max(a),min(b) FROM T SAMPLE BY 1h FILL(PREV);
-- 2023-01-01T01:00:00.000000Z  max1  min1 --
-- 2023-01-01T02:00:00.000000Z  max1  min1 (filled)
-- 2023-01-01T03:00:00.000000Z  max3  min3 --
```

Listing 2. New Feature in QuestDB (Time-Series)

for relational database systems [5, 24, 30, 41–43, 57], uncovering numerous previously unknown bugs and demonstrating the effectiveness of their designed oracles.

Graph database systems such as Neo4J [37] and TinkerPop [4] have seen rapid growth over the past decade, driven by their native support for graph-structured data. Despite their relative recent emergence compared with relational systems, there is already substantial work on testing these engines [19, 22, 26, 32, 46, 56, 58]. Given these extensive efforts in proposing effective testing approaches, we do not further focus on graph database systems in this paper.

Emerging database systems refer to database engines introduced within the past decade that target specialized workloads and exhibit sustained adoption growth (*e.g.,* an average increase exceeding 10%). Beyond graph databases, rapidly growing categories include time-series, vector, streaming, wide-column, and spatial database systems. These engines provide dedicated features that expand functionality beyond traditional relational settings (*e.g.,* native vector similarity, windowed time-series operators, or geospatial predicates). In contrast to their widening adoption, automated bug discovery for such systems remains comparatively underexplored [50, 52].

To compare popularity trajectories, we collect statistics from the above database survey platforms [15, 16] and summarize each system's popularity trends from 2013. Figure 1 summarizes trends for widely used systems over the past decade: graph database systems show the steepest rise, time-series systems are among the fastest-growing non-graph categories, and relational systems remain relatively stable as a mature baseline. While emerging database systems have seen rapid growth, the development of robust testing methodologies for them has not kept pace with the attention devoted to established relational and graph database systems.

Q2: New features. To effectively showcase the landscape of emerging database systems, we have opted to focus on two types: (i) time-series database systems due to their growing popularity shown in Figure 1, and (ii) streaming database systems for their increasing importance in time-critical tasks. Rather than enumerating all their features, below, we focus on explaining some important core features to give an intuition of their differences from relational database systems.

Time-series database systems have gained prominence in managing data points indexed by time, making them ideal for applications like Internet of Things (IoT) devices, financial trading, and monitoring systems. These database systems (*e.g.,* QuestDB [39] and TDEngine [48]) provide new features (*e.g.,* date arithmetic and time-series downsampling) related to date and timestamp as well as storage optimization and retrieval of timestamp data, enabling efficient analysis and visualization of trends over time. Listing 2 gives an example query in QuestDB, which utilizes special clauses, namely sample by and fill, to visualize the histogram of timestamp data. This query retrieves results from the table, identifying maximum and minimum values based on timestamps sampled at one-hour intervals. Additionally, it fills in the results based on the preceding one in case no results are available for a given duration. Another new feature is the clause latest on, which retrieves the most recent entry by timestamp for a given key or combination of keys. Additionally, time-series database systems can include various timestamp functions or operations like dateadd() and datediff(). Such new features provide a direct way to access or modify timestamped data.

Streaming database systems like RisingWave [44] are vital for applications demanding real-time data updates and continuous synchronization across users and systems. These systems are used in applications that need streaming features, such as messaging platforms, collaborative tools, and live

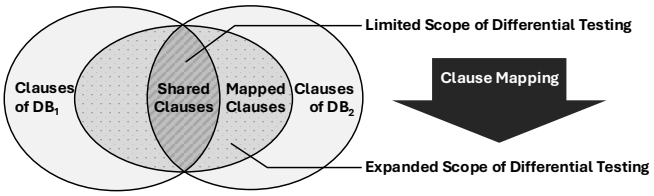


Fig. 2. How ACME Improves Differential Testing

analytics dashboards, which require low-latency access to data and concurrent user interactions. To achieve this goal, streaming database systems introduce various new features. For instance, RisingWave provides a new window function called tumble. The time window refers to the temporal intervals that users can utilize to segment events and execute data computations in the streaming process. The tumble(table_or_source, start_time, window_size) function in RisingWave generates a new table alternative for data selection. Additionally, it supports window join, which combines several time windows for convenient queries in streaming data.

Q3: Query languages. In light of varied data models proposed in various emerging database systems, statistics from the *database of databases* [15] show a total of 18 distinct query interfaces (*e.g.,* command line, SQL, custom API, Cypher, Datalog). Among 450 individual database systems that have surfaced over the past decade, 158 instances (35.1%) have opted for SQL-like queries as their interfaces. This places SQL-like query languages as the most common choice. Hence, this work focuses on testing emerging database systems using SQL as a reference point.

SQL clauses serve as the fundamental building blocks of SQL queries. An SQL query is typically composed of multiple *clauses*, each specifying a particular component of the query. For example, the SELECT clause defines the attributes to retrieve, the FROM clause specifies the source tables, and the WHERE clause restricts the result set with predicates. Other clauses, such as GROUP BY, HAVING, and ORDER BY, control aggregation and result organization. Differences in their syntax or semantics across database systems will lead to differences in their results.

3 Approach

Our insight is that many emerging database systems conceptually extend traditional relational models. This potentially allows logic bugs to be identified by comparing their query results with those from mature relational systems. However, a challenge is that there are intrinsic differences between the emerging database systems and relational ones, both in query syntax as well as their semantics (*e.g.,* features such as new kinds of SQL-like clauses). Thus, regular differential testing may be ineffective with either high false positives or limited coverage.

Intuition. Figure 2 illustrates the core idea of ACME. Consider two SQL-like database systems: $DBMS_1$, a traditional relational database system, and $DBMS_2$, an emerging database system. While some queries are directly comparable because their clause syntax and semantics agree (we call these *shared clauses*), others contain differences, which we call *non-shared clauses*. To expand the scope of differential testing, we introduce *clause mappings*, which transform clauses with differing syntax or semantics into SQL-equivalent forms, increasing the set of queries that can be tested. We highlight that completeness of clause mappings is not needed; we found partial mappings provide substantial benefits. In practice, as few as 5–20 mappings can significantly enhance query diversity and coverage, making our approach scalable and applicable across a variety of database systems.

Approach overview. ACME consists of two tightly connected phases. The first phase is an *LLM reasoning phase*, where we leverage large language models (*e.g.,* GPT-4o) to analyze the syntax

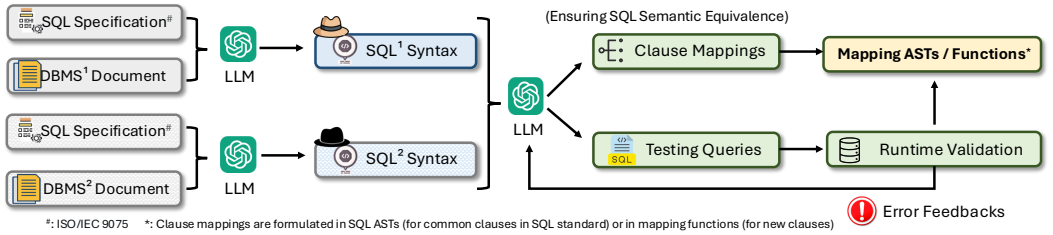


Fig. 3. ACME Workflow for Generating Clause Mappings

and semantics of SQL and SQL-like queries across different database systems. Through structured prompts, the LLM constructs candidate clause mappings. These rules encapsulate the programmatic logic required to translate dialect-specific syntax while strictly preserving the underlying query semantics. The second phase is an *enhanced differential testing phase*, where these rules are applied to generate semantically equivalent, but syntactically varied query pairs. This significantly broadens the scope of differential testing: if two queries are semantically equivalent, their results should remain consistent across the emerging database system under test and a relational reference system.

Figure 3 illustrates the core workflow of ACME for generating clause mappings via LLMs. The process begins by identifying the SQL specifications (*i.e.,* ISO/IEC 9075 [20]) and documentation of the target DBMSs.[5] ACME leverages the LLM to propose candidate clause mappings, which serve as bridges between the standard SQL dialect of a reference relational database system and the specialized query language of the emerging database system. The candidate clause mappings are tested and if the validation testing phase passes, these clause mappings are implemented as either *AST-based visitors* for systematic tree rewrites or *mapping functions* in Python for flexible, context-aware translations. The system and user prompts for generating mappings are as follows:

System Prompt

You are a Database Engineer analyzing SQL result mismatches between **{DBMS_OLD}** and **{DBMS_NEW}**.

User Prompt for Clause Mapping

Task: Write a Python mapping function that translates QuestDB queries to PostgreSQL queries to resolve syntax errors or semantic mismatches.

Context:
QuestDB SQL query (source): `{type_marker}{questdb_query}{type_marker}`
PostgreSQL query error: `{type_marker}{error_messages}{type_marker}`
Last failed translation: `{type_marker}{error_mapped_postgres_query}{type_marker}`

Requirements:

(1) Output a Python script.
(2) The script **must** define a function `def transform(query):` that takes a SQL string and returns the corrected SQL string.
(3) You may use the `re` module and/or string replacement.
(4) Also provide a `validation_query` containing the expected correct PostgreSQL query.

Output JSON only:

```
{
  "mappings": [
    {
```

[5]Given the rapid advancement of LLM capabilities, system specifications and documentation are typically either already integrated into their training data or easily retrieved via web searches.

```
      "filename": "{rule_filename}",
      "code": "import re\n\ndef transform(query):\n  # logic here\n    return query"
    }
  ],
  "validation_query": "CORRECTED_PG_QUERY"
}
```

To ensure that these mappings preserve semantics, we incorporate a *mapping query validation* step, which validates that queries generated under different syntax remain semantically equivalent through executions on both relational and emerging database systems. Any failures or inconsistencies are fed back to the LLM for iterative refinement, where ACME re-prompts LLMs for patched mappings with the error messages and retries until the correct validation or the maximum attempt limit is reached. By validating semantic consistency through execution, ACME significantly improves the robustness of generated clause mappings and reduces false positives in differential testing.[6] Although some incorrect mappings may initially pass the validation phase and later surface as false alarms in corner cases, ACME re-examines these cases and retries the mappings with the additional feedback. This closed-loop design ensures that even partial mappings add value by increasing query diversity and coverage while progressively enhancing the robustness of differential testing.

Listing 3 illustrates one example of a generated clause mapping expressed in the SQLGlot [34] AST format. In this instance, the mapping addresses the semantic divergence in how QuestDB and PostgreSQL handle NULL elements within IN predicates. By transforming the predicate into a CASE expression, the rule explicitly enforces three-valued logic: it ensures that a NULL subject correctly yields a NULL result, while isolating the membership check to non-null elements. This not only prevents false alarms arising from inherent semantic divergence but also facilitates the detection of logic bugs within these specialized clauses. Generated clause mappings are applied during query generation to produce semantically equivalent queries across varied SQL dialects.

Query generator and test oracle. ACME creates pairs of SQL queries for relational and emerging database systems by randomly selecting and combining valid clauses. Clause mapping ASTs are then applied to transform these queries into differential input pairs for both relational and emerging database systems. The outputs of the paired queries are compared, and any difference indicates a logic bug in the emerging database system.

Clause mapping examples for QuestDB. We illustrate ACME using a time-series database as a concrete example of query mapping. From the statistics in Figure 1, time-series database systems are the second-fastest-increasing database model in the past decade. To test them, we first create a database schema with a timestamp column consistently set as the primary key for the table. We

```
# Original: (E IN L), e.g., (c0 IN (0, NULL))
exp.In(this=exp.Column(this=exp.Identifier(this="E")),
       expressions=[exp.Identifier(this="L_item"), exp.Null()])

# Mapped: (CASE WHEN E IS NULL THEN NULL ELSE E IN L_nonnull END)
# e.g., (CASE WHEN c0 IS NULL THEN NULL ELSE c0 IN (0) END)
exp.Case(
    ifs=[exp.If(
        this=exp.Is(this=exp.Column(this=exp.Identifier(this="E")),
                    expression=exp.Null()),
        true=exp.Null())],
    default=exp.In(this=exp.Column(this=exp.Identifier(this="E")),
                   expressions=[exp.Identifier(this="L_item")]))
```

Listing 3. Example of SQL AST Mapping between QuestDB and PostgreSQL

[6]We remark that we only test for equivalence in the context of certain query transformations and executions.

Table 1. Illustrative List of Simplified Clause Mappings from QuestDB to PostgreSQL

ID	Clause	QuestDB	PostgreSQL Mapping Result
01	*Sample By*	*select count(*) from T sample by 1h*	*select .. (select .. extract(hour from date) as b from T group by b)*
02	*Null*	*select A in (1,2,3,null)*	*select case when A is null then null else A in (1,2,3) end*
03	*DateAdd*	*select dateadd('h',1,ts)*	*select cast((cast(ts as integer)+3600) as timestamp)*
04	*DateDiff*	*select datediff('y',now(),now())*	*select abs(extract(year from now())-extract(year from now()))*
05	*Latest On*	*select * from t latest on a partition by b*	*.. (select .. join (select distinct max(a) over(partition by b) ..*
06	*Distinct*	*select count_distinct(c0) from test*	*select count(distinct c0) from test*
07	*Symbol*	*create table test (c0 int, c1 symbol, c2 timestamp)*	*create table test (c0 int, c1 varchar(128), c2 timestamp)*
08	*Between*	*select count(*) from T where c0 between 2 and 0*	*select count(*) from T where c0 symmetric between 2 and 0*

demonstrate clause mappings for QuestDB, a fast-growing time-series database system that follows SQL syntax. Similar mappings exist for other time-series database systems. PostgreSQL is used as a reference for a reliable relational database system because it is one of the most popular and well-known relational database systems. In Table 1, we present notable clause mappings along with corresponding query examples before and after the mappings. A detailed explanation follows.

(i) Mapping for clause sample by. From Section 2, sample by is a new clause introduced in QuestDB to summarize large datasets into aggregates of homogeneous time periods in a select statement (*e.g.,* to process histogram of timestamp data). Consider the query select count(*) from sensors sample by 1h in QuestDB, which retrieves record numbers every hour. Relational database systems, however, lack support for such clauses. Thus, we translate these clauses into the following SQL query: select res from (select count(*) as res, extract(hour from date) as hour from sensors group by hour).

(ii) Mapping for null values. The implementation of null in QuestDB differs from the SQL standard: QuestDB treats it as a distinct value, whereas SQL regards it as a marker rather than a data point. When a comparison involves null, QuestDB outputs boolean values while PostgreSQL outputs null. To bridge this gap, we make use of the shared clause case...when to first identify the null value in comparison operands, and if it exists, outputs null directly in QuestDB instead of boolean values. Consider the query select A in (B). We map this query into select case when A is null then null else A in (B) end to align the results with PostgreSQL. Similar clause mappings exist in other comparison clauses (*e.g.,* between). While this appears simple, it is effective in reducing the false alarms of differential testing.

(iii) Mapping for date operations including dateadd, datediff, and in. These functions provide more straightforward interfaces for directly operating on timestamp data. For the dateadd, we first cast timestamp data to an integer and cast it back after calculations. For example, dateadd('h',1,ts) is translated into cast((cast(ts as integer)+3600) as timestamp). For datediff, the query datediff('y',now(),now()) in QuestDB returns the result 0, indicating that the two dates differ by zero years. We adopt clause mappings in our approach and translate it into abs(extract(year from now())-extract(year from now())) as a valid and semantically equivalent query in PostgreSQL. The in operator checks whether the given data is within the range. With timestamp data, it can be used as ts in '2023' to check if the timestamp is in the year 2023. We adapt it to relational database systems via the clause between. For example, ts in '2023-01;-3d' can be translated to ts between '2023-01-01 00:00:00' and '2023-01-28 23:59:59'.

(iv) Mapping for clause latest on. The latest on clause in QuestDB retrieves the most recent data row by timestamp for given keys. Consider the query select * from t latest on c0 partition by c1 where c0 represents the designated timestamp primary key and c1 is an integer data column. PostgreSQL does not support this feature. We map it into select distinct * from (select t1.c0,t1.c1,t1.c2,t1.c3 from t as t1 join (select distinct max(c0) over(partition by c1) as c0, c1 from t) as t2 on t1.c0=t2.c0 and t1.c1=t2.c1) latest_on by replacing the latest on with table joins and a semantically equivalent window function.

(v) In addition to dedicated features in emerging database systems, clause mappings are essential when common clauses have different semantics or syntax in two database instances. For example, the clauses between and symmetric between are both used to select values within a given range (*i.e.,* between the lower and upper bounds, inclusively). Some database systems, like QuestDB, implicitly use symmetric between to accommodate invalid ranges, while PostgreSQL explicitly requires the symmetric keyword. Another crucial aspect of clause mapping is type aliasing, which is necessary when working with various database systems that use different type names. To ensure consistency and prevent syntax errors, we employ data type alias mappings including symbol to varchar or text, int to integer, long to bigint, and timestamp to datetime. In some cases, clause mappings help connect similar clause usages. For example, QuestDB has special functions for distinct aggregation, such as count_distinct(x), while PostgreSQL uses the distinct keyword inside the count function as count(distinct x). Although these clause mappings may seem straightforward, they are vital for accurate differential testing across dissimilar database systems.

Clause mappings for streaming database systems. Streaming database systems are designed to handle continuous data with specialized optimizations and advanced features. Unlike traditional relational systems, they process data immediately upon arrival rather than after it has been fully stored. We illustrate representative clause mappings in RisingWave [44] as follows: (i) Mapping tumble. In RisingWave, the tumble function produces a derived table annotated with time window boundaries. Since relational database systems do not support this construct directly, we emulate it using sub-queries with additional columns window_start and window_end. For example, SELECT * FROM tumble(t, c0, INTERVAL '1 day') can be translated into PostgreSQL as: SELECT * FROM (SELECT *, date_trunc('day', c0) AS window_start, date_trunc('day', c0) + interval '1 day' AS window_end FROM t ORDER BY window_start, window_end) AS tumble_table. (ii) Mapping hop. The hop function is similar to tumble but generates overlapping hop windows. We apply the same translation strategy as above, adjusting intervals to reflect the hop size and slide. (iii) Mapping window join. A window join combines tuples across windows, either with base tables or other windows of the same type. Since both tumble and hop are mapped into sub-queries in (i) and (ii), their windowed outputs can be joined directly in PostgreSQL using equality on keys and alignment of window boundaries.

The clause mappings illustrated above highlight several recurring patterns that arise when adapting emerging database systems for differential testing. First, mappings operate at different levels of abstraction: some address surface-level syntactic differences (*e.g.,* type aliases, function renaming), while others encode deeper semantic divergences that require structural query rewrites (*e.g.,* null handling, windowed aggregation). Second, even simple mappings such as type aliasing are load-bearing. Without them, queries fail at schema initialization before any meaningful testing can occur. Third, mappings serve a dual purpose: suppressing false alarms from legitimate semantic differences while exposing genuine bugs when discrepancies persist after mapping. These patterns recur across both time-series and streaming systems, suggesting the approach generalizes beyond any single database system category. Together, the diversity and subtlety of these mappings motivate leveraging LLMs for their automated discovery, as manual enumeration does not scale.

4 Implementation

We briefly introduce some implementation details not covered in Section 3.

Database creation. Before each round of testing, we initialize three tables. Each table can have up to 8 columns, assigned data types such as integer, string, or timestamp, and is populated with 50 to 500 rows of randomly generated data, with special values (*e.g.,* null) taken into consideration. For time-series database systems, we additionally designate a timestamp column as the primary key, as required by their query semantics (*e.g.,* LATEST ON in QuestDB references the designated timestamp primary key).

Query generating strategies. We generate SQL statements based on the three initialized tables by randomly combining shared and mapped clauses derived from our approach. Our query generation is modular and scalable, allowing users to enable or disable individual clauses or features as needed. We use several strategies to improve the complexity of generated queries: (i) replacing tables or expressions with sub-queries, an important SQL feature that increases structural complexity; (ii) using query concatenations (*i.e.,* UNION, EXCEPT, and INTERSECT) to combine multiple queries into a single, more complex query; and (iii) incorporating advanced features such as window functions and branching expressions.

Result analysis. The test oracle of `ACME` expects that semantically equivalent query pairs produce identical results across the emerging and relational database systems. Any discrepancy flags a potential logic bug. Beyond logic bugs, `ACME` also detects internal errors by inspecting unexpected exceptions. We leverage heuristics to identify unexpected exceptions (*e.g.,* NullPointerException, OutOfBoundException) while filtering out valid ones (*e.g.,* feature-not-supported errors).

5 Evaluation

In this section, we answer the following questions to assess various important aspects of `ACME`:

- **Q1 Discovery of Unknown Bugs**. How effective is `ACME` in discovering previously unknown bugs in popular emerging database systems? (Section 5.1)
- **Q2 Improvement on Differential Testing**. To what extent does `ACME` enhance the effectiveness of differential testing? (Section 5.2)
- **Q3 Comparison with Existing Techniques**. What is the improvement in testing emerging database systems compared with the state-of-the-art database testing approaches? (Section 5.3)
- **Q4 Effectiveness and Feasibility of LLM-generated Clause Mappings.** How effective and feasible are LLM-generated clause mappings in testing emerging database systems? (Section 5.4)

Selected database systems. To assess the efficacy of our approach, we have selected a subset of emerging SQL-like database systems, namely QuestDB [39] (16.1K stars), TDEngine [48] (24.3K stars), RisingWave [44] (8.3K stars), and CrateDB [14] (4.3K stars). Emerging database systems have a diverse range of types and models as shown in Section 2, making it challenging to cover all of them in this work. The selection is based on their increasing popularity and similarity to standard SQL syntax. We deliberately avoid including emerging database systems that are implemented as close extensions of well-known relational ones (*e.g.,* Timescale [49], closely mirrors PostgreSQL syntax, thus differential testing is trivially applicable). For relational database systems, we leverage PostgreSQL [38] as the reference ground truth for testing. Our evaluation focuses primarily on QuestDB, which differs substantially from PostgreSQL in both its newly introduced features and semantic variations. To demonstrate the broader applicability of `ACME` to other emerging SQL-like database systems, we also applied it to TDEngine, CrateDB, and RisingWave. In these systems, we identified 6 bugs in TDEngine, 4 bugs in RisingWave, and 3 bugs in CrateDB, although we have applied comparatively less testing effort than for QuestDB.

Experimental setup. We performed all experiments on a personal computer with Intel(R) Core(TM) i7-14700 CPU, 32 GB RAM, and NVIDIA GeForce RTX 2060 6 GB. The OS is Ubuntu 20.04.2 LTS. `ACME` can detect all confirmed or fixed bugs within six hours of running on personal computing resources.

5.1 Discovering Unknown Bugs

To detect new logic bugs in emerging database systems, we intermittently tested the latest versions of the target emerging database systems, using PostgreSQL-14.0 to derive the ground-truth results, over a period of three months, which is a typical methodology for evaluating the effectiveness

Table 2. Unknown Bugs Found by ACME

Database System	Internal Errors			Logic Bugs			Total
	Pending	Confirmed	Fixed	Pending	Confirmed	Fixed	
QuestDB	0	0	34	1	2	9	**44**
TDEngine	0	1	2	1	1	1	**6**
RisingWave	0	0	3	0	1	0	**4**
CrateDB	0	0	2	0	0	1	**3**
Total	**0**	**1**	**41**	**2**	**4**	**11**	**59**

of automatic testing tools [26, 43]. We choose PostgreSQL as the reference relational database system because it is widely adopted, actively maintained, and compliant with the SQL standard. Its mature implementation and extensive documentation make it a reliable baseline for interpreting clause semantics. Moreover, PostgreSQL has been widely used as a reference system in prior testing research [30, 41–43] and shown to be the least buggy, ensuring continuity and comparability with existing work. In most cases, ACME took a few hours to find a bug with only small computing resources. We reported bugs after reducing bug-inducing queries and checking whether the issue had already been reported on issue trackers to avoid duplicate bug reports. Automatically generated bug-inducing test cases are typically complex, and we reduced them to a smaller bug-inducing version by delta debugging [55]. Next, we illustrate bugs with reduced queries, omitting unnecessary query mappings.

Results. Table 2 summarizes the number of unknown bugs identified using our approach. We classified the identified bugs into two distinct categories: *(i) Internal Errors* refer to bugs where a query caused unexpected aborts, exceptions, or crashes in the target database system; *(ii) Logic Bugs* refer to bugs found through discrepancies flagged by the differential testing. Additionally, we categorized bugs into three statuses: *(i) Pending* bugs are those that have been identified and submitted, but are awaiting further investigation by developers to determine the root cause; *(ii) Confirmed* bugs are those that have been acknowledged by developers but have not yet been fixed; *(iii) Fixed* bugs are those that have been confirmed and subsequently fixed by the developers.

In total, we identified 59 unknown bugs (17 logic bugs and 42 internal errors), of which 5 were confirmed and 52 were fixed. For illustration, we present parts of noteworthy bugs that ACME identified, describing them based on their root cause through our analysis and developers' feedback.

Logic bug—incorrect string comparison. A common feature of database systems is string comparisons, which are commonly supported by either using specific functions like strcmp in MySQL or directly via operators (*e.g.,* >,<,=). In QuestDB, we observed one discrepancy[7] that occurred when counting distinct results with string comparison in the predicate, as shown in Listing 4 from PostgreSQL. To trigger this bug, ACME generates differential inputs with varying

```
CREATE TABLE test (c_0 SYMBOL);
INSERT INTO test VALUES ('A');
SELECT count_distinct(c_0) FROM test WHERE c_0>'Z';
-- Emerging Database System Result: [(1)] --
CREATE TABLE test (c_0 VARCHAR(16));
INSERT INTO test VALUES ('A');
SELECT count(DISTINCT c_0) FROM test WHERE c_0>'Z';
-- Relational Database System Result: [(0)] --
```

Listing 4. Incorrect String Comparison in QuestDB

[7] https://github.com/questdb/questdb/issues/3828

syntax by utilizing clause mappings. These rules map specific clauses, such as symbol to varchar or the count operator, to string-type keywords. This example illustrates the simplicity of clause mapping and the enhanced effectiveness it brings to differential testing.

Logic bug—incorrect set clauses. The intersect and except clauses are commonly used to combine results from two sub-queries. We found a bug[8], shown in Listing 5, which concerns the logic of intersect and except clauses in QuestDB. Another similar bug was found when executing complex SQL structures with multiple joins and the same column names in CrateDB, as shown in Listing 5. We observe the difference in the result set from PostgreSQL after the necessary type mapping (*e.g.,* to cast timestamp into bigint for comparison) when executing union queries with duplicated output columns. These bugs were resolved after we submitted the cases causing the bug to the developers.

Logic bug—incorrect nested joins and window function. Complex query structures (*e.g.,* with nested joins or window functions) tend to trigger bugs in emerging database systems. One bug **ACME** found is related to nested joins. A nested join in SQL involves the use of multiple join operations within a single query, which allows for the retrieval of data that spans multiple tables. Listing 6 shows Q1, a bug-inducing query computing an incorrect result due to an unknown bug.[9] The issue is in a transitive filter pass when handling nested table joins. Another bug that **ACME** found is related to the window function, which calculates its results across a set of rows related to the current row. Unlike regular aggregate functions, window functions do not collapse the result set into a single value for each group. In QuestDB, we observe one bug-inducing case,[10] shown as Q2 in Listing 6, which outputs differently compared to PostgreSQL when executing window function queries with order by clause and table joins.

Internal errors—incorrect syntax errors. As one category of internal errors, **ACME** also detects various incorrect syntax errors through expanded differential testing. These bugs were found by observing syntax errors in QuestDB while the same queries returned correctly in PostgreSQL. As shown in Listing 7, the query Q1 or Q2 reports invalid column c0/c1 in QuestDB,[11] whereas the PostgreSQL version executes normally.

Internal errors—core dumped/out of bound. Another notable category of bugs we encountered involves internal errors where the target database systems crashed. As shown in Listing 8, **ACME** found a segmentation fault in TDEngine[12] and an out-of-bound error in CrateDB.[13] To date, 42 internal errors have all been confirmed or fixed by the developers.

```
(SELECT 1 UNION ALL SELECT 1) EXCEPT (SELECT 0);
(SELECT 1 UNION ALL SELECT 1) INTERSECT (SELECT 1);
-- QuestDB: [(1),(1)]   PostgreSQL: [(1)] --
CREATE TABLE test (c_0 TIMESTAMP, c_1 INT, c_2 FLOAT);
INSERT INTO test VALUES (946702800000, 8, 3.0);
INSERT INTO test VALUES (946688400000, 9, 7.0);
INSERT INTO test VALUES (946695600000, 6, 8.0);
(SELECT .. FROM test as t1 CROSS JOIN ...) UNION (...);
-- CrateDB: [(3,1,1)..]   PostgreSQL: [(3,2,1)..] --
```

Listing 5. Incorrect Set Clauses in QuestDB/CrateDB

[8] https://github.com/questdb/questdb/issues/3580
[9] https://github.com/questdb/questdb/issues/4010
[10] https://github.com/questdb/questdb/issues/3936
[11] https://github.com/questdb/questdb/issues/3933
[12] https://github.com/taosdata/TDengine/issues/22857
[13] https://github.com/crate/crate/issues/14805

```
CREATE TABLE t(c0 TIMESTAMP, c1 INT, c2 INT);
INSERT INTO t VALUES('2025-01-01␣10:00:00', 1, 1);
INSERT INTO t VALUES('2025-01-01␣10:00:00', 1, 1);
Q1: SELECT count(1) FROM t as T1 CROSS JOIN t as T2 WHERE T2.c0='2000-01-01␣
    00:00:00' INTERSECT SELECT count(1) FROM t as T1 JOIN t as T2 on T1.c0=T2.c0
    JOIN t as T3 ON T2.c0=T3.c0;
-- QuestDB Result: [(0)] PostgreSQL Result: [] --
Q2: SELECT avg(T1.c_1) OVER(PARTITION BY 1=1 ORDER BY T1.c_0) FROM t AS T1 CROSS
    JOIN t AS T2;
-- QuestDB: [(1),(1.33)] PostgreSQL: [(1),(1.5)] --
```

Listing 6. Logic Bugs with Nested Joins or Window Functions

```
CREATE TABLE test (c0 short, c1 timestamp); -- QuestDB
CREATE TABLE test (c0 float, c1 timestamp); -- PostgreSQL
Q1: SELECT avg(c0) FROM test UNION SELECT DISTINCT avg(c0) FROM test;
-- QuestDB: "Invalid Column c0" PostgreSQL: [()] --
Q2: SELECT count(1) FROM test AS T1 JOIN test AS T2 ON T1.c1<T2.c1 JOIN test AS T3
     ON T2.c1=T3.c1;
-- QuestDB: "Invalid Column c1" PostgreSQL: [()] --
```

Listing 7. Unexpected Syntax Errors in QuestDB

```
CREATE TABLE test(c0 TIMESTAMP);
INSERT INTO test VALUES ('2025-01-01␣00:00:00.000');
Q1: SELECT count(1) FROM test AS a JOIN test as b ON a.c0=b.c0 AND a.c0 IS NULL;
-- TDEngine: FATAL crash signal...(core dumped) --
Q2: SELECT DISTINCT count ... sys.summits as T2 JOIN sys.summits as T3 ON T2.
    mountain>T3.mountain;
-- CrateDB: Index 2 out of bounds --
```

Listing 8. Internal Errors in TDEngine/CrateDB

```
SELECT count(1) FROM telemetry WHERE telemetry.event>(telemetry.event IN ());
-- QuestDB:ArrayOutOfBoundsException Postgres:[(1)] --
SELECT * FROM test WHERE SUM("", "");
-- QuestDB:StringOutOfBoundsException Postgres:[(1)] --
SELECT count(1) FROM test as a INNER JOIN test as b ON a.time=b.time WHERE now()=
    now();
-- QuestDB:NullPointerException Postgres:[(1)] --
```

Listing 9. Unexpected Exceptions in QuestDB

Unexpected exceptions in QuestDB which output normally in PostgreSQL. Listing 9 shows bugs leading to the following unexpected exceptions: *ArrayIndexOutofBoundsException*,[14] *StringIndexOutofBoundsException*,[15] *NullPointerException*.[16] In PostgreSQL, these queries are valid and produce expected results.

5.2 Improvement on Differential Testing

One key question is how much we improve upon traditional differential testing. We evaluate the enhanced effectiveness of our differential testing approach on the following three metrics and compare ACME with and without clause mapping: (i) discovering more bugs, (ii) covering more unique query plans, and (iii) achieving higher query execution success rates.

Enhanced effectiveness in detecting logic bugs. ACME identifies significantly more logic bugs, benefiting from the clause mapping technique, which broadens the scope of differential testing by

[14] https://github.com/questdb/questdb/issues/3469
[15] https://github.com/questdb/questdb/issues/3467
[16] https://github.com/questdb/questdb/issues/3324

bridging non-shared clauses in the two target database systems. To evaluate the impact of clause mappings, we first conducted a manual analysis of all 17 logic bugs identified by ACME to determine the necessity of these mappings for triggering the bugs. Following this, we compared the clause success rates with and without the use of clause mappings.

Among 17 logic bugs identified by ACME, 7 of them are detected with the assistance of clause mappings. The necessary clause mappings for reproducing logic bugs include (i) type mappings, as type keywords may vary across different database instances (*e.g.,* the string type may be represented as symbol or varchar). Without type mappings, ACME will miss most identified bugs as test cases (*e.g.,* Listing 4) would fail at the create table statement; (ii) function mappings, where functions like count_distinct() in emerging database systems require rewriting as count(distinct) for correct functionality; and (iii) feature mappings, which align new features with valid keywords in relational database systems as introduced in Section 3. However, not every logic bug necessitates reproduction via clause mappings. For instance, the inaccurate intersect/except results in Listing 5 are deemed valid in both emerging and relational database systems even without the aid of mappings. Our methodology expands differential testing by bridging syntax gaps and mapping new features, allowing many SQL queries to be valid in both systems.

Covering more unique query plans. A query plan specifies the ordered steps (*e.g.,* table scans, table joins) that database systems will take to access data. Created by the query optimizer, it aims to find the most efficient way to execute the query based on factors such as the database schema, data distribution, and available indexes. A higher number of executed query plans increases the probability that the testing tool can uncover unknown bugs, as demonstrated in existing works [5, 6].

We conducted a 24-hour experiment using ACME to compare the unique query plans covered by classic differential testing (*i.e.,* without clause mapping) and an expanded scope of differential testing (*i.e.,* ACME, with clause mapping). As shown in Figure 4, ACME covers 14,707 additional unique query plans, representing an improvement of approximately 20% after 24 hours of testing. These results suggest that the expanded scope of differential testing improves the ability to execute a greater variety of query plans, thereby uncovering more unknown issues compared to traditional differential testing.

Higher query execution success rate. To investigate the impact of clause mappings on differential testing across both emerging and relational database systems, we compare query execution success rates (*i.e.,* queries that complete without error) with and without clause mappings. The success rates of queries exhibit significant variations under different configurations. We noted instances where the success rate dropped to 0% when type mappings were not enabled, resulting in failures during table initialization. The success rate remained stable when new features were infrequently incorporated into query generations. In summary, we observed a decrease in query success

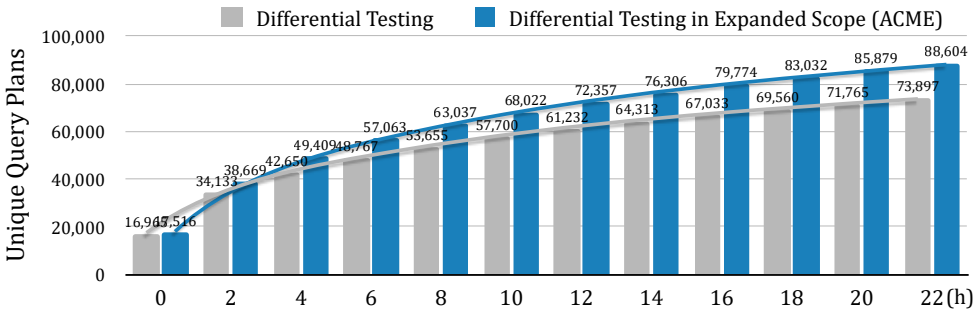


Fig. 4. ACME's Improvement on Unique Query Plans via Clause Mappings

```
-- Bug: QuestDB Result [] vs. PostgreSQL [(1)]--
(SELECT DISTINCT avg(c0) over(partition by 1) FROM t);
-- TLP-True Query QuestDB Result [] --
(SELECT .. over(partition by 1) FROM t WHERE c0>0);
-- TLP-False Query QuestDB Result [] --
(SELECT .. over(partition by 1) FROM t WHERE not c0>0);
-- TLP-Null Query QuestDB Result [] --
(SELECT .. over(...) FROM t WHERE c0>0 is NULL);
```

Listing 10. Partitioning with TLP is not effective to detect QuestDB bug

rates ranging from around 23.9% (*e.g.,* only queries with new features output syntax errors) to 100% (*e.g.,* all queries failed without necessary type mappings in create table statements) on QuestDB, RisingWave, CrateDB, and PostgreSQL. This highlights the importance of clause mappings in enhancing the effectiveness of differential testing across heterogeneous database systems.

5.3 Comparison with Existing Techniques

We now evaluate the effectiveness of ACME compared with existing techniques. To the best of our knowledge, there is only one (non-public) existing work on testing emerging database systems, Unicorn [50], which focuses on fuzzing time-series database systems. Unicorn proposed a hybrid input generation to create valid SQL queries for time-series database systems. However, as Unicorn uses a crash oracle, it is unable to detect more complex bugs like unexpected syntax errors and logic bugs.

To further assess ACME, we ported the state-of-the-art test oracle for relational database systems, Ternary Logic Partition (TLP) [42] implemented in SQLancer [43], to our selected emerging database systems. [17] This was necessary, as TLP is not officially implemented for emerging database systems. Different from our approach, TLP leverages metamorphic relations that partition one query into three sub-queries, each selecting rows based on whether a predicate generates true, false, or null. We compare the effectiveness by checking how many unknown logic bugs ACME found can be detected by using the TLP test oracle. Then, we evaluate ACME against SQLancer by comparing unique query plans and code coverage, demonstrating our improvements in testing emerging database systems.

Effectiveness. We used the same methodology as prior works [41, 42], that is, to conduct a manual and best-effort analysis to identify bugs found by ACME that TLP overlooked. To adapt TLP for emerging database systems, we modified TLP to partition the original queries as a manual best-effort process, leveraging existing predicates or introducing relevant predicates that refer to columns in the original queries in cases where no predicates exist.

For 17 bug-inducing test cases to which test oracles could be applied, TLP failed to reveal any of these bugs, showing the strength of our differential testing approach across database systems. The primary reason why TLP fails to detect them is that the presence of predicates does not influence the incorrect outcome. The root causes of such bugs are traced back to other clauses, such as over(partition by), join, union, or unrelated expressions like null and features. In instances of these bugs, the addition of predicate partitions does not impact the results. A specific example is illustrated in Listing 10. The bug was initially found using differential testing, using PostgreSQL as a reference, which revealed a discrepancy within a sub-query. Upon query reduction, the bug-inducing scenario

[17] Reasons for using TLP include: TLP is the state-of-the-art test oracle for finding logic bugs in SQL database systems. TLP can also find most bugs that (NoREC) [41] finds. Query Plan Guidance (QPG) [5] is a test-case generation approach, and not a test oracle. Pivoted Query Synthesis (PQS) [43] is no longer maintained in SQLancer and thus not used for comparison. Differential Query Execution (DQE) [46] aims at testing Data Manipulation Language (DML) statements, which are out of the scope of this paper.

Table 3. Code Coverage (# of Covered Instructions) Comparison with SQLancer

	io.questdb.cairo				Overall			
Tool	**10 m**	**1 h**	**10 h**	**1 d**	**10 m**	**1 h**	**10 h**	**1 d**
SQLancer	16,572	17,074	17,122	17,122	77,200	80,047	80,385	80,411
ACME	19,689	19,863	19,863	19,863	83,222	83,459	83,717	83,834

reveals that introducing brackets to a query with a window function causes the bug. Although we add predicates related to the sole column reference in the query, TLP does not influence the result and fails to find this bug.

In addition to logic bugs, 18 of 42 internal errors ACME found due to inconsistent aborts. Detecting such bugs requires specialized parsing of output error messages and typically necessitates additional handlers for aborts or exceptions (*e.g.*,, Unicorn [50]), which may require further manual effort. In contrast, our enhanced differential testing approach leverages the correct output of the reference database system, making it easy to identify such bugs.

Coverage. Code coverage is a widely used metric to evaluate the effectiveness of database system testing approaches [42, 43]. We evaluate the instruction coverage of ACME and SQLancer on QuestDB over 24 hours (as suggested in [27]) using the publicly available JaCoCo [21] library. We report two statistics: overall coverage and focused coverage on the SQL engine (*io.questdb.cairo*), which excludes components unrelated to query processing, such as connection handling and authentication.

Table 3 shows that ACME achieves higher coverage than SQLancer in both metrics throughout the 24-hour period. Both tools saturate quickly on the SQL engine (SQLancer reaches 17,122 instructions by 10h and ACME reaches 19,863 by 1h), reflecting that even state-of-the-art testing tools face challenges in fully supporting emerging database systems, as SQLancer's generated queries for QuestDB are limited to a small number of simple query structures. ACME maintains a higher focused and overall coverage across the entire testing period.

5.4 LLM Reliability and Feasibility

To evaluate the efficacy of using LLMs for clause mapping, we benchmark ACME against two baselines: (1) *no mappings*, where PostgreSQL rejects QuestDB queries outright due to syntax and type incompatibilities; and (2) *essential mappings*, which apply manual heuristics for basic type conversions and NULL handling (*i.e.,* the #02 and #07 clause mappings from Table 1). We report two metrics over 1,000 mapping queries: the *QuestDB (QDB) success rate* (*i.e.,* the percentage of queries accepted by QuestDB), which remains stable at ∼72–73% across all configurations as it is not affected by clause mapping; and the *PostgreSQL (PG) success rate* (*i.e.,* the percentage of mapped queries accepted by PostgreSQL), which reflects mapping quality. We additionally report token consumption under 10- and 50-attempt budgets as a proxy for practical cost. Furthermore, we conduct an ablation study to evaluate the impact of test-query validation and re-prompting. We focus specifically on a lightweight model, as its higher susceptibility to errors makes iterative refinement particularly critical for achieving correct mapping results.

LLM effectiveness. As shown in Table 4, essential mappings raise the PostgreSQL success rate from 0% to 35.5%, short of practical testing needs (query generation combines clauses to obtain a test query). Integrating LLMs into ACME yields large gains: GPT-5.2 reaches 64.8% at 10 attempts and 69.9% at 50 attempts, nearly doubling the heuristic baseline, while even the smallest model (Qwen2.5-Coder-7B) surpasses the heuristic at 50 attempts (51.5%). Increasing the attempt budget

Table 4. PostgreSQL (PG) query success rate after clause mapping from QuestDB, and token usage, over 1,000 queries. (Δ) denotes improvement over the *Essentials* baseline. The QuestDB success rate remains stable at ~72–73% across all configurations and is not affected by clause mapping.

		PG Success Rate		Tokens (K)	
Category	**Model**	**10 Attempts**	**50 Attempts**	**10 Attempts**	**50 Attempts**
Baselines	No-mappings	0%		–	
	Essentials (type+null)	35.5%		–	
LLMs	GPT-5.2	**64.8%** (+29.3)	**69.9%** (+34.4)	7.5	33.2
	GPT-4o	56.0% (+20.5)	60.0% (+24.5)	10.0	41.8
	Qwen3-Coder-30B	40.6% (+5.1)	62.9% (+27.4)	10.2	60.4
	Qwen2.5-Coder-7B	36.6% (+1.1)	51.5% (+16.0)	20.7	75.5

consistently improves performance across all models, most notably for Qwen3-Coder-30B (+22.3%), though at the cost of greater token consumption.

LLM feasibility. `ACME` is designed to operate effectively without exclusive reliance on high-parameter, proprietary models. The cross-model evaluation spans four representative computational tiers: GPT-5.2 (flagship), GPT-4o (high-performance), Qwen3-Coder-30B (medium-scale open-weight), and Qwen2.5-Coder-7B (lightweight open-weight). Results demonstrate that mid-tier open-weight models such as Qwen3-Coder-30B (62.9% at 50 attempts) can achieve competitive performance relative to GPT-4o (60.0%), at a fraction of the API cost since open-weight models can be run using local compute resources. Although smaller models consume more tokens to arrive at valid mappings (*e.g.,* 75.5K tokens for Qwen2.5-Coder-7B versus 33.2K for GPT-5.2 at 50 attempts, owing to more frequent re-validation cycles), they still achieve meaningful improvements over the heuristic baseline (+16.0% for Qwen2.5-Coder-7B). These results confirm that `ACME` is broadly applicable across deployment environments, enabling effective differential testing locally without prohibitive token costs or dependence on online LLM APIs.

LLM reliability. We further conduct an ablation study examining the contribution of query validation and re-prompting. Using Qwen2.5-Coder-7B as the subject model (lightweight models being more prone can generate more syntactically incorrect mappings), we assess the impact of bypassing validation entirely (*i.e.,* accepting all LLM-generated mappings without the validation step). Disabling both validation and re-prompting causes the PostgreSQL query success rate to drop from 42.9% to 34.8%, falling below even the heuristic baseline; this degradation is primarily due to invalid SQL syntax in unvalidated mappings. In contrast, retaining the validation pipeline allows even lightweight models to consistently exceed the heuristic baseline, confirming the contribution of the re-prompting to ensuring mapping precision.

6 Discussion

We now discuss three important issues from our approach.

Manual effort. We initially designed clause mappings manually; however, our current approach can automatically discover them with the help of LLMs, requiring only minimal manual effort. `ACME` leverages LLMs' extensive knowledge of SQL syntax and semantics, and also for the emerging database systems. Although LLMs can automate clause mappings, we still require some human effort in query generation, result analysis, and validation of complex clause mappings. Most existing database system testing approaches also require similar manual effort, often due to subtle differences in SQL dialects. For example, SQLancer [43] implements about 3,000 lines of DBMS-specific Java

code to handle syntax and semantic variations, while Thanos [18] manually collects supported features across storage engines to enable differential testing.

LLM reliability. ACME incorporates runtime validation to mitigate incorrect clause mappings produced by LLMs. Nevertheless, some fragile mappings may persist, especially when interacting with untested clauses, requiring ACME to re-prompt the LLM for alternative mappings. While certain mappings may remain infeasible even after multiple attempts, completeness is not necessary for effective bug finding. In practice, we find that obtaining a few valid mappings (*e.g.,* 10 rounds of interaction in Table 4) is sufficient to significantly expand differential testing coverage, incurring only minimal token costs in LLM usage.

False alarm. We encountered only 3 false alarms out of 55 confirmed bugs while developing and running ACME. We addressed them by fixing or removing the clause mapping. This is a common methodology which has been used in various differential testing approaches [9, 13, 25, 56]. While LLM-generated clause mappings can introduce additional false alarms in differential testing, these are largely mitigated by ACME's validation process; however, manual inspection is still necessary to maintain a minimal false-alarm rate, particularly when using small-to-medium-scale models.

Prompt engineering. A core insight of ACME is leveraging mature database system clauses to construct semantically equivalent rewrites for features in emerging database systems, with LLMs automating this mapping process. However, the prompts currently used in ACME are empirically effective but not optimized. Their performance may vary across different database system versions, LLM updates, and query contexts. More principled prompt engineering techniques, such as few-shot examples, chain-of-thought reasoning, or retrieval-augmented generation, could be integrated into ACME to further improve the reliability and correctness of LLM-generated clause mappings.

7 Related Work

We briefly summarize the most relevant research directions.

Detecting logic bugs in database systems. Logic bugs (incorrect query results returned by DBMSs) are notoriously difficult to detect, because they require a reliable *test oracle*, a mechanism for deciding whether a query result is correct. Song *et al.* proposed *Differential Query Execution* (DQE) [46], which detects logic bugs by checking whether SQL queries with equivalent predicates access the same rows. Rigger and Su introduced several metamorphic oracles for relational databases, including *Pivoted Query Synthesis* (PQS) [43], *Non-Optimizing Reference Engine Construction* (NoREC) [41], and *Ternary Logic Partitioning* (TLP) [42], which have uncovered hundreds of bugs. To support these oracles, SQLRight [30] employs coverage-guided test generation, while *Query Plan Guidance* (QPG) [5] steers tests toward unseen query plans to diversify behaviors. In contrast to these works, which focus primarily on relational databases, we address the more challenging task of finding bugs in *emerging databases* that differ not only from relational systems but also from one another. This is fundamentally harder than checking consistency across systems that all implement the same relational model and SQL standard.

Detecting performance issues. DBMSs are sensitive to performance regressions. Benchmark suites such as TPC-H [1] expose inefficiencies on standardized workloads. APOLLO [25] detects regressions by comparing execution times of the same query across different DBMS versions, while AMOEBA [31] identifies slowdowns by executing semantically equivalent query pairs.

Differential testing. Differential testing [35] detects bugs by executing the same input across multiple implementations of the same specification. This approach has been widely applied in domains such as web services [9], JVM implementations [13], compilers and interpreters [23, 53], and network protocols [7]. For database systems, *RAGS* [45] compared query results across different systems, while APOLLO [25] applied differential testing to execution times for regression detection.

Our work extends this line of research by making differential testing scalable to emerging database systems, where syntax and semantic divergences make direct comparison difficult.

Metamorphic testing. Metamorphic testing [10, 12] is another powerful oracle-based technique that generates semantically-related test pairs and validates whether a *metamorphic relation* holds between their outputs. It has been successfully applied in bioinformatics [11, 40], graph database systems [22], web services [8], embedded systems [28], Datalog engines [33], compilers [29, 51], and neural networks [54]. We compare `ACME` against TLP implemented in SQLancer [42], the state-of-the-art metamorphic oracle for relational database systems.

Detecting memory errors in database systems. Much prior work on testing DBMSs targets memory errors, which do not require an explicit oracle. Grey-box fuzzers such as AFL [2] use mutation and coverage guidance, but naive mutation often produces invalid SQL inputs. To address this, Squirrel [57] enforces syntax-preserving mutation, while grammar-based generators such as SQLSmith [3], DynSQL [24], and ADUSA [31] produce valid queries by construction. Griffin [17] further avoids grammar engineering by adopting a grammar-free generation strategy. Although these approaches generate test cases without syntax errors, they do not address the oracle problem and thus cannot reliably detect logic bugs.

8 Conclusion

Testing emerging database systems for logic and other bugs remains a largely unexplored area, where the primary challenge lies in developing a scalable testing approach. The core insight of our work is that many emerging systems can be viewed as extensions of relational database systems, sharing substantial functionality that enables cross-system testing. Building on this observation, we have proposed an enhanced differential testing framework that leverages relational systems as a reliable ground truth and addresses key challenges via three steps: (i) constructing clause-mapping requests to LLMs on the fly whenever differential testing encounters unsuccessful queries, (ii) validating the generated clause mappings by executing test queries and re-prompting upon validation failures, and (iii) generating semantically equivalent yet syntactically diverse queries to broaden the coverage of differential testing. Applying `ACME` to four widely used emerging database systems, we discovered 59 previously unknown bugs, including 17 logic bugs and 42 internal errors, with 57 confirmed or fixed by vendors. Beyond bug discovery, `ACME` discovers more bugs and achieves greater code coverage than existing tools. Furthermore, our experiments confirm that `ACME` remains effective even under a lightweight LLM budget using local models, making it a practical solution for continuous integration. The real-world utility of our work is underscored by acknowledgments from industry developers, such as QuestDB, highlighting the effectiveness of LLM-driven reasoning in building a unified, scalable, and robust testing framework for the next generation of new database systems.

Acknowledgments

We thank the maintainers of the evaluated database systems for their prompt responses and for their efforts in addressing the issues we identified. This research is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing <NRF-NCR25-Fuzz-0001>) and by MOE grant A-8001544-00-00. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] 1988. TPC-H Benchmark. https://www.tpc.org/tpch/.

[2] 2013. American Fuzzy Lop (AFL) Fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[3] Sjoerd Mullender Andreas Seltenreich, Bo Tang. 2025. https://github.com/anse1/sqlsmith.

[4] Apache. 2025. https://tinkerpop.apache.org/.

[5] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2060–2071. https://doi.org/10.1109/ICSE48619.2023.00174

[6] Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proc. ACM Manag. Data* 2, 3, Article 188 (May 2024), 26 pages. https://doi.org/10.1145/3654991

[7] David Brumley, Juan Caballero, Zhenkai Liang, and James Newsome. 2007. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *16th USENIX Security Symposium (USENIX Security 07)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/16th-usenix-security-symposium/towards-automatic-discovery-deviations-binary

[8] Wing Kwong Chan, Shing Chi Cheung, and Karl RPH Leung. 2007. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research (IJWSR)* 4, 2 (2007), 61–81.

[9] Peter Chapman and David Evans. 2011. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '11)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/2046707.2046737

[10] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases*. Technical Report HKUST-CS98-01. Department of Computer Science, Hong Kong.

[11] Tsong Yueh Chen, Joshua WK Ho, Huai Liu, and Xiaoyuan Xie. 2009. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC bioinformatics* 10, 1 (2009), 1–12.

[12] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1, Article 4 (jan 2018), 27 pages. https://doi.org/10.1145/3143561

[13] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 85–99. https://doi.org/10.1145/2908080.2908095

[14] CrateDB. 2025. https://cratedb.com/.

[15] dbdb. 2025. https://dbdb.io/.

[16] DBEngines. 2025. https://db-engines.com/.

[17] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2023. Griffin: Grammar-Free DBMS Fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 49, 12 pages. https://doi.org/10.1145/3551349.3560431

[18] Ying Fu, Zhiyong Wu, Yuanliang Zhang, Jie Liang, Jingzhou Fu, Yu Jiang, Shanshan Li, and Xiangke Liao. 2025. *Thanos: DBMS Bug Detection via Storage Engine Rotation Based Differential Testing*. IEEE Press, 655–666. https://doi.org/10.1109/ICSE55347.2025.00001

[19] Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, Wenpin Jiao, and Tao Xie. 2023. GDsmith: Detecting Bugs in Cypher Graph Database Engines. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/3597926.3598046

[20] International Organization for Standardization. 2016. ISO/IEC 9075: Information technology — Database languages — SQL. Available from https://www.iso.org/standard/63555.html.

[21] Jacoco. 2025. https://www.jacoco.org/.

[22] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland H. C. Yap, Zhenkai Liang, and Manuel Rigger. 2024. Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 46, 12 pages. https://doi.org/10.1145/3597503.3623307

[23] Yuancheng Jiang, Jianing Wang, Qiange Liu, Yeqi Fu, Jian Mao, Roland HC Yap, and Zhenkai Liang. 2025. ZendDiff: Differential Testing of PHP Interpreter. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1095–1106. https://doi.org/10.1145/3533767.3534409

[24] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, 4949–4965. https://www.usenix.org/conference/usenixsecurity23/presentation/jiang-zu-ming

[25] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: automatic detection and diagnosis of performance regressions in database systems. *Proc. VLDB Endow.* 13, 1 (sep 2019), 57–70. https://doi.org/10.14778/3357377.3357382

[26] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. 2023. Testing Graph Database Engines via Query Partitioning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 140–149. https://doi.org/10.1145/3597926.3598044

[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[28] Fei-Ching Kuo, Tsong Yueh Chen, and Wing K. Tam. 2011. Testing embedded software by metamorphic testing: A wireless metering system case study. In *2011 IEEE 36th Conference on Local Computer Networks*. 291–294. https://doi.org/10.1109/LCN.2011.6115306

[29] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *SIGPLAN Not.* 49, 6 (jun 2014), 216–226. https://doi.org/10.1145/2666356.2594334

[30] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4309–4326. https://www.usenix.org/conference/usenixsecurity22/presentation/liang

[31] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 225–236. https://doi.org/10.1145/3510003.3510093

[32] Qiuyang Mang, Aoyang Fang, Boxi Yu, Hanfei Chen, and Pinjia He. 2024. Testing Graph Database Systems via Equivalent Query Rewriting. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 143, 12 pages. https://doi.org/10.1145/3597503.3639200

[33] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic testing of Datalog engines. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 639–650. https://doi.org/10.1145/3468264.3468573

[34] Tobias Mao and contributors. 2023. *SQLGlot: A Python SQL Parser and Transpiler.* https://github.com/tobymao/sqlglot https://github.com/tobymao/sqlglot.

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

[36] MySQL. 2025. https://www.mysql.com/.

[37] Neo4j Graph Platform. 2025. https://neo4j.com/.

[38] PostgreSQL. 2025. https://www.postgresql.org/.

[39] QuestDB. 2025. https://questdb.io/.

[40] Arvind Ramanathan, Chad A. Steed, and Laura L. Pullum. 2012. Verification of Compartmental Epidemiological Models Using Metamorphic Testing, Model Checking and Visual Analytics. In *2012 ASE/IEEE International Conference on BioMedical Computing (BioMedCom)*. 68–73. https://doi.org/10.1109/BioMedCom.2012.18

[41] 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1140–1152. https://doi.org/10.1145/3368089.3409710

[42] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (nov 2020), 30 pages. https://doi.org/10.1145/3428279

[43] 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)*. USENIX Association, 667–682. https://www.usenix.org/conference/osdi20/presentation/rigger

[44] RisingWave. 2025. https://www.risingwave.com/.

[45] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 618–622.

[46] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing Database Systems via Differential Query Execution. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2072–2084. https://doi.org/10.1109/ICSE48619.2023.00175

[47] SQLite. 2025. https://www.sqlite.org/.

[48] TDEngine. 2025. https://tdengine.com/.

[49] Timescale. 2025. https://www.timescale.com/.
[50] 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* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 251–262. https://doi.org/10.1145/3533767.3534364
[51] Dongwei Xiao, Zhibo LIU, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic Testing of Deep Learning Compilers. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1, Article 15 (feb 2022), 28 pages. https://doi.org/10.1145/3508035
[52] Rui Yang, Yingying Zheng, Lei Tang, Wensheng Dou, Wei Wang, and Jun Wei. 2023. Randomized Differential Testing of RDF Stores. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 136–140. https://doi.org/10.1109/ICSE-Companion58688.2023.00041
[53] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532
[54] Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2023. Unveiling Hidden DNN Defects with Decision-Based Metamorphic Testing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 113, 13 pages. https://doi.org/10.1145/3551349.3561157
[55] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498
[56] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. 2022. Finding bugs in Gremlin-based graph database systems via Randomized differential testing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 302–313. https://doi.org/10.1145/3533767.3534409
[57] 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* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 955–970. https://doi.org/10.1145/3372297.3417260
[58] Zeyang Zhuang, Penghui Li, Pingchuan Ma, Wei Meng, and Shuai Wang. 2024. Testing Graph Database Systems via Graph-Aware Metamorphic Relations. *Proc. VLDB Endow.* 17, 4 (mar 2024), 836–848. https://doi.org/10.14778/3636218.3636236

Received 2025-09-12; accepted 2025-12-22