One DBMS, Two Modes, and a Bunch of Bugs: Catching Logic Bugs in Distributed DBMSs via Differential Testing

ZI-XUAN FU, Beihang University, China
JIA-JU BAI*, Beihang University, China
HONG-BO FENG, Beihang University, China
KANG CHEN, Tsinghua University, China

Logic bugs in distributed database management systems silently yield incorrect results and are difficult to detect, yet they threaten correctness in critical deployments. Existing testing techniques focus mainly on system level failures or general DBMS and do not target distributed execution, leaving many distributed logic bugs undiscovered. We present DistSQL, a differential testing framework that compares the results of identical SQL queries executed on the same DBMS configured in centralized and distributed modes. The key insight is that centralized execution is simpler and typically better tested, and thus can serve as a practical reference for distributed execution. DistSQL addresses two difficulties that hinder effective bug finding in distributed settings: it performs distributed diversity oriented database state mutation to expose distribution specific behaviors, and it conducts distributed interaction guided exploration of the execution space using query plan features to prioritize novel behaviors over redundant tests. DistSQL requires no intrusive code instrumentation.

The evaluation on five popular open source distributed DBMSs, including TiDB, CockroachDB, YugabyteDB, ClickHouse, and OceanBase, as well as one widely deployed commercial DBMS, demonstrates the efficacy of DistSQL. DistSQL identified 65 previously unknown logic bugs, including 38 specific to distributed execution. Of these, 61 have been confirmed and 50 have been fixed.

CCS Concepts: • **Information systems → Database query processing**; **Relational parallel and distributed DBMSs**; • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Distributed DBMS testing, logic bugs

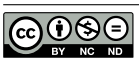
ACM Reference Format:
Zi-Xuan Fu, Jia-Ju Bai, Hong-Bo Feng, and Kang Chen. 2026. One DBMS, Two Modes, and a Bunch of Bugs: Catching Logic Bugs in Distributed DBMSs via Differential Testing. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 59 (February 2026), 24 pages. https://doi.org/10.1145/3786673

1 Introduction

Distributed database management systems (*i.e.*, distributed DBMSs) are widely deployed in critical domains [10, 60]. By distributing data and processing logic across multiple computing nodes, distributed DBMSs significantly enhance application performance and scalability. Logic bugs in distributed DBMSs (as well as in single node DBMSs) are particularly critical. Unlike system level issues such as memory bugs [1, 22, 65] or performance bugs [8, 32], which typically manifest as

*Jia-Ju Bai is the corresponding author.

Authors' Contact Information: Zi-Xuan Fu, fzx24@buaa.edu.cn, Beihang University, Beijing, China; Jia-Ju Bai, baijiaju@buaa.edu.cn, Beihang University, Beijing, China; Hong-Bo Feng, fenghongbo@buaa.edu.cn, Beihang University, Beijing, China; Kang Chen, chenkang@tsinghua.edu.cn, Tsinghua University, Beijing, China.



ACM 2836-6573/2026/2-ART59
https://doi.org/10.1145/3786673

Fig. 1. A simplified example: the same query returns inconsistent results under centralized and distributed execution.

```
INSERT INTO t1 (c1,c2,c3) VALUES
  (1, false, 0), (2, false, 0), (3, true, 0);
SELECT DISTINCT
  first_value(c1) over (PARTITION BY c2 ORDER BY c1)
FROM t1;
```

crashes or performance anomalies, logic bugs silently produce incorrect results. These subtle errors are difficult for users to detect and can propagate through applications, leading to unexpected and potentially severe consequences. As such, effective tools for detecting logic bugs are crucial for ensuring the correctness and reliability of distributed DBMSs.

Motivating example. Figure 1 presents a simplified example (details in § 7.5). Since `c2` has only two distinct values, the window function should yield at most two distinct `first_value(c1)` results; however, ClickHouse [9] returns three rows under distributed execution (three nodes) but the expected result on a single node. According to the developers, this is caused by parallel window-function execution, highlighting that distributed execution can introduce extra execution paths and logic bugs.

However, to the best of our knowledge, no tool is designed specifically for detecting logic bugs in distributed DBMSs.

(1) Bug detection for distributed systems: Several existing approaches target system level bugs in distributed systems, such as concurrency and error handling bugs. For example, some approaches [34, 58] find concurrency bugs by exploring different execution interleavings. Other approaches detect error handling bugs by injecting faults [15, 26, 35, 52]. A few general purpose bug detection approaches [20, 36] can find logic bugs in distributed DBMSs, but they rely on fixed SQL workloads. Because these tools are designed primarily to expose system level issues, they are less effective at identifying logic bugs that involve richer SQL semantics.

(2) Bug detection for general DBMSs: Numerous approaches detect logic bugs in DBMSs by validating against predefined test oracles and can be applied to both centralized and distributed systems. Some approaches check the consistency of semantically equivalent predicates or expressions across multiple statements [2, 18, 24, 44, 45, 48, 61]. Others verify the correctness of query optimization [43, 53] or specific features such as indexes and transactions [3, 23, 30, 49]. However, these approaches do not target distributed execution and are therefore less effective at identifying distributed logic bugs.

To this end, we design and implement the DistSQL framework to detect logic bugs in distributed DBMSs. DistSQL rests on two observations. *(1) Many distributed DBMSs support both centralized and distributed execution modes. (2) If the same query produces different results across modes, this indicates at least one bug in the database.* Because the centralized mode is easier to deploy and is often used for testing and debugging, it is generally better tested and more reliable than the distributed mode. Leveraging this, DistSQL performs differential testing by comparing results of identical SQL queries under both modes. Specifically, we deploy two instances of the same DBMS, one in centralized mode (single node) and the other in distributed mode (multiple nodes), execute the same SQL queries on both, and compare their outputs to identify inconsistencies.

Although implementing the framework is straightforward, efficiently exposing bugs in the distributed mode is difficult because distributed execution poses additional challenges absent in the centralized mode.

Challenge 1 (C1): How should we mutate the database state so that it more effectively exposes distributed logic bugs? Unlike centralized systems, distributed databases have a far more diverse

and complex state space due to partitioning, replication, and inter-node coordination. In principle, these states should not affect the correctness of SQL results. However, flawed systems may exhibit anomalous behavior under particular states. Naïve or purely random state mutations rarely cover this diversity, leading to repetitive tests and overlooking many potential distributed scenarios. Therefore, it is crucial to design a state-mutation method that captures the system's distributed characteristics, generating a variety of meaningful distributed states and thereby exercising more distributed execution paths.

Challenge 2 (C2): How can we efficiently explore the explosive execution space to uncover distributed logic bugs? The collaborative behaviors intrinsic to distributed DBMSs, such as inter-node data exchange and task coordination, substantially increase the scale and complexity of the execution space. Here, "execution space" denotes the set of all possible executions of a query, encompassing alternative execution plans and scheduling strategies. Efficient exploration requires determining which regions of the execution space are covered by each test case. This insight enables removal of redundant or ineffective tests and focuses testing on meaningful distributed behaviors.

In principle, DistSQL aims, at both the database state and execution space levels, to trigger diverse execution behaviors so that bugs causing divergent results are exposed more efficiently. DistSQL introduces two techniques to address these challenges.

Distributed diversity oriented database state mutation. During testing, we either generate data or modify existing data (i.e., perform database state mutation). DistSQL designs state mutation algorithms to broaden coverage of states specific to distributed DBMSs. For example, we vary partitioning schemes, replication factors, and data placement. These mutations induce richer cross node interactions and distributed processing, exercising states introduced by distribution, which are absent in centralized settings and hard for prior methods to reach, thereby revealing distributed specific logic bugs that manifest under such conditions.

Distributed interaction guided execution space exploration. As in fuzzing testing, efficient exploration relies on ***feedback*** to prioritize new execution behaviors over rerunning already covered logic. Because DistSQL targets logic peculiar to distributed systems, it centers on operator interactions, namely how an operator obtains data from its children and propagates results to its ancestors, as operations may run in different nodes. Concretely, ***feedback*** used by DistSQL represents a query plan's interaction *feature* (detailed in § 5.1) as a tuple `<pattern, score>`, where *pattern* denotes the operator subgraph that participates in the interaction and *score* quantifies its relevance to distributed execution. We consider two patterns in prose: an operator together with all of its children, which captures how inputs are aggregated, and an operator together with its parent and grandparent, which captures upward propagation, including propagation that indirectly reaches the grandparent.

Using these two techniques, DistSQL achieves two goals: (1) for database states that struggle to trigger new logic, we keep mutating them until they do so efficiently; and (2) for queries that already trigger new logic, we continue mutating them to further expose related distributed behaviors. DistSQL comprises three components: a distributed state mutator designed to cover diverse distributed scenarios; a runtime monitor with feature pools that extract and track query plan features to guide both test case generation and database state mutation; and a differential checker that compares query results between centralized and distributed executions and flags mismatches as potential logic bugs.

This paper makes three primary contributions:

- We show that differential testing between centralized and distributed execution serves as an effective test oracle for distributed DBMSs. Using the simpler and typically better tested logic of

Table 1. Reported issue types across four distributed DBMSs.

DBMS	logic	consistency	memory	assert	settings	status	perf.	funcmiss	resource	stats	enhance	others	Total
ClickHouse [9]	**365**	78	186	257	168	58	59	400	32	37	6	40	1686
CockroachDB [11]	**89**	138	70	63	128	46	82	133	39	96	42	71	997
TiDB [41]	**409**	136	186	61	96	66	83	198	41	166	14	77	1533
YugaByteDB [59]	**131**	306	198	104	99	94	105	139	60	102	68	87	1493
Total	**994**	658	640	485	491	264	329	870	172	401	130	275	5709
Ratio	**17.4%**	11.5%	11.2%	8.5%	8.6%	4.6%	5.8%	15.2%	3.0%	7.0%	2.3%	4.8%	100%

Table 2. Comparison of state-of-the-art testing approaches.

Approach	*Main Target*	*Bug Type*	*Feedback*	*Black-Box*	*Logic Checker*	*Database State Mutation*
Morpheus [58]	Distributed System	Concurrency	Conflict Analysis	No	-	-
Oathkeeper [33]	Distributed System	Semantic	None	No	-	-
Chronos [8]	Distributed System	Timeout	None	No	-	Generic
Jepsen [20]	Distributed System	Logic	None	Yes	Fixed Rules	Fixed Input
Mallory [36]	Distributed System	Logic	Timeline Summary	No	Fixed Rules	Fixed Input
RAGS [47]	DBMS	Logic	None	Yes	Differential	Generic
Squirrel [65]	DBMS	Memory	Branch Coverage	No	-	Generic
Lego [28]	DBMS	Memory	Type-Affinity	No	-	Generic
CODDTest [61]	DBMS	Logic	None	Yes	Metamorphic	Generic
EET [24]	DBMS	Logic	None	Yes	Metamorphic	Generic
DQP [3]	DBMS	Logic	None	Yes	Metamorphic	Generic
Mozi [29]	DBMS	Logic	None	Yes	Differential	Generic
Radar [49]	DBMS	Logic	None	Yes	Differential	Generic
SemConT [31]	DBMS	Logic	Keyword and Rule	Yes	Verification	Generic
TQS [53]	DBMS	Logic	Graph Similarity	Yes	Ground Truth	Generic
QPG [2]	DBMS	Logic	Query Plan Text	Yes	Metamorphic	Generic
DistSQL (this work)	Distributed DBMS	Logic	Query Plan Feature	Yes	Differential	Favor Distributed Logic

centralized execution as a reference, our approach finds distributed logic bugs often missed by existing techniques.

- We develop DistSQL, which integrates two techniques: (1) *distributed diversity oriented database state mutation, and (2)* distributed interaction guided execution space exploration. These techniques address two core challenges in testing distributed DBMSs by effectively triggering diverse distributed executions which can efficiently explore the explosive execution space.
- We evaluate DistSQL on five widely used open source distributed DBMSs, TiDB [41], CockroachDB [11], YugabyteDB [59], ClickHouse [9], and OceanBase [37], as well as a widely deployed commercial DBMS. In total, DistSQL finds 65 previously unknown logic bugs across these six systems, including 38 that manifest only in distributed execution and 27 that appear in both centralized and distributed execution. Among them, 61 have been confirmed and 50 have already been fixed. These results highlight DistSQL's effectiveness and its advantage over existing techniques in detecting logic bugs in distributed DBMSs.

2 Background and Motivation

2.1 Logic Bugs in Distributed DBMSs

To assess the prevalence of logic bugs in real-world distributed DBMSs, we analyzed issues from four widely used open source distributed DBMSs with active development communities. We collected issues reported between March 2024 and March 2025 and classified them in three steps: (1) keyword-based grouping over tags, titles, and descriptions to form coarse categories; (2) prompting a large language model with definitions and examples of twelve bug types and then let the model assign the most relevant category to each issue; and (3) manual review and correction. Issues from CI/CD systems that lacked meaningful context were excluded.

Finally, we categorized these issues into twelve types: (1) *logic*: incorrect SQL results; (2) *consistency*: state or data inconsistency; (3) *memory*: buffer overflows, null pointer dereferences, and similar defects; (4) *assert*: assertion failures; (5) *settings*: configuration changes not taking effect; (6) *status*: unexpected system status; (7) *perf*: performance degradation; (8) *funcmiss*: missing functionality, expected functionality unsupported or unavailable; (9) *resource*: resource exhaustion; (10) *stats*: inaccurate statistics; (11) *enhance*: enhancement requests; and (12) other issues. Categories (1) through (6) concern correctness and reliability, while (7) through (12) cover performance, usability, and other noncritical reports.

Among these categories, logic bugs are the most prevalent, accounting for 994 of 5,709 issues (17.4%), exceeding the shares of memory bugs (11.2%) and assertion failures (8.5%). This prevalence underscores that logic bugs are a critical and urgent challenge in distributed DBMSs.

2.2 Limitations of Existing Approaches

We survey representative approaches from distributed system testing that involve DBMSs as part of the system under test, along with several classical approaches to DBMS testing. A selection of these approaches is summarized in Table 2.

Distributed system testing. Morpheus [58] finds concurrency bugs by exploring different interleavings to identify inconsistent system states. OathKeeper [33] checks for semantic violations arising from broader system operations such as configuration changes or administrative commands, but its notion of semantics differs from SQL correctness. Chronos [8] focuses on timeout bugs, validating whether systems behave as expected under timeout conditions. These approaches target bug categories that are fundamentally different from logic bugs in DBMSs.

Jepsen [20] finds logic bugs in distributed DBMSs by executing a fixed workload under concurrent operations and fault injection. It verifies key correctness properties that distributed systems are expected to guarantee, such as isolation levels and linearizability, and has found various bugs in practice. However, its focus on specific constraints of distributed systems and manually defined verification rules limits its ability to validate broader SQL semantics of DBMSs, making it challenging to capture the full semantic richness of real-world DBMS behavior. Mallory [36] builds on Jepsen by improving the efficiency of fault injection, but it inherits the same limitations in terms of SQL semantic coverage and workload generality.

In summary, while some distributed system testing approaches target different classes of problems, those that verify distributed logic tend to focus on specific distributed constraints and pay less attention to diverse SQL semantics, which in turn narrows the scope of distributed logic testing for distributed DBMSs.

DBMS testing. RAGS [47] performs differential testing across DBMS implementations to find logic bugs. However, it is constrained by the limited set of features shared among DBMSs, which narrows the scope of its testing. Squirrel [65] and Lego [28] targets memory-related bugs, which are out of scope for this work.

Several approaches aim to find logic bugs in DBMSs through unguided test case generation. EET [24], and CODDTest [61] find logic bugs by rewriting queries into another forms that are expected to produce the same results. However, the test cases are limited by the scope of these transformation rules. Mozi [29] and Radar [49] find logic bugs by executing the same statements under different internal options or table constraints and comparing results. DQP [3] compares executions with different query hints applied, where query plans act as local heuristics rather than guiding test case generation. These three methods tend to find bugs tied to specific features but does not test other aspects of database processing logic.

Some approaches leverage specific scenario feedback to guide test case generation. SemConT [31] formalizes SQL semantics and verifies results against a formal model, using rule coverage to guide generation. However, its formal model struggles with features such as correlated subqueries and diverse expressions. TQS [53] splits a wide table into multiple smaller tables and joins them to test the correctness of join operations, guiding generation by test case similarity. Similarly, it cannot generate statements with subqueries as join operands or non-equality joins.

QPG [2] improves SQLancer [50] by using query plan text as feedback to mutate the database state when no new plans are found, which focuses only on centralized database states and ignores distributed ones. Although this approach is effective under low-diversity workloads, its discriminative power decreases as SQL diversity increases. In our evaluation on TiDB, 5 million queries produced over 2.3 million unique plans (47.93%), revealing low feedback granularity. Consequently, QPG is less effective on distributed DBMSs, as its feedback signal becomes noisy and less informative.

Overall, while these approaches have shown effectiveness in finding certain classes of logic bugs, their testing scope is inherently limited by their methodology. In addition, their database mutation strategies do not account for distributed database states, such as data distribution across nodes. Moreover, even when applied to distributed DBMSs, their focus on a single execution mode prevents them from capturing inconsistencies specifically introduced by distributed execution, and it also prevents them from developing specialized runtime feedback and mutation strategies, leading to missed distributed logic bugs.

2.3 Motivation and Challenges

DistSQL rests on two observations. *(1) Many distributed DBMSs support both centralized and distributed execution modes. (2) If the same query produces different results across modes, this indicates at least one bug in the database.* Because the centralized mode is easier to deploy and is often used for testing and debugging, it is generally better tested and more reliable than the distributed mode. Motivated by this insight, our differential testing approach compares DBMS behavior in these two modes to identify potential logic bugs. Although aimed at distributed logic bugs, this method also has the potential to find bugs that manifest in the centralized mode. This is because different results only indicate that the DBMS may have a latent bug. They do not, by themselves, prove that it arises from the distributed mode. As discussed above, however, the likelihood of triggering bugs under distributed conditions is higher. Because both modes are supported, a separate reference model as an oracle is unnecessary. Building such a reference model for each DBMS is burdensome due to semantic differences across systems and the potential performance overhead on large queries.

Although conceptually simple, applying differential testing to distributed DBMSs introduces additional challenges.

Challenge 1 (C1): How should we mutate the database state so that it more effectively exposes distributed logic bugs? Unlike centralized systems, distributed databases have a much more diverse and complex state space due to partitioning, replication, and inter-node coordination. For example, in centralized mode a table is local, whereas in distributed mode the same table can be partitioned across nodes in many ways (e.g., split across A and B, or entirely on C), greatly enlarging the state space. However, running the same SQL in both modes often fails to produce different behavior, because naïve or purely random state mutations frequently do not exercise distributed execution. Without sufficient database state diversity, the distributed mode may cover only a narrow subset of behaviors and thus resemble the centralized mode, thereby reducing the effectiveness of differential testing.

Challenge 2 (C2): How can we efficiently explore the explosive execution space to uncover distributed logic bugs? Compared to single node systems, distributed DBMSs introduce inter-node communication, coordination, and parallel execution, greatly expanding the execution space. For

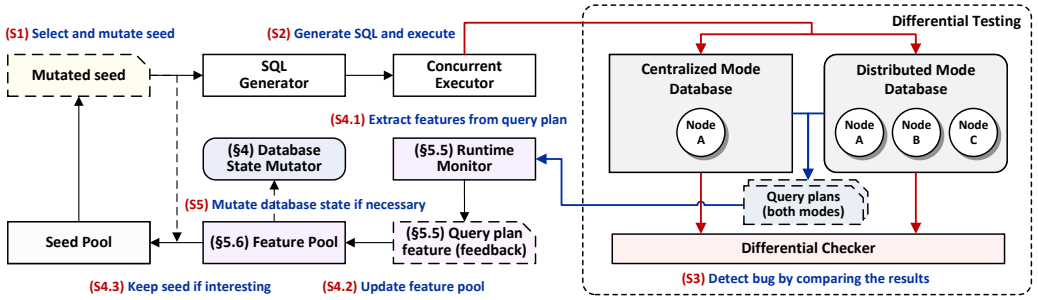


Fig. 2. DistSQL workflow. **Database State Mutator(§ 4):** Mutates the database state during initialization or to explore more diverse distributed behaviors. **SQL Generator(§ 3.2):** generates new SQL statements from a given seed. **Concurrent Executor:** executes batches of test cases in parallel on both centralized and distributed modes to enable differential comparison. **Runtime Monitor(§ 5.5):** Collects query plans from the DBMS and extracts query plan features as feedback that characterize each query's execution behavior. **Feature Pool(§ 5.6):** Tracks all observed query plan features and marks a test case as interesting when it exercises previously unseen distributed logic. **Seed Pool(§ 6):** Maintains seeds for interesting test cases and mutates them to progressively explore more regions of the execution space. **Differential Checker:** Compares results under centralized and distributed modes for the same query and flags any *mismatch*.

example, in centralized mode a UNION runs locally, whereas in distributed mode it may (1) push data to remote nodes, (2) pull data to the local node, or (3) partially process remotely, leading to a much larger execution space. Existing approaches often lack informative feedback or rely on weak signals that fail to indicate which regions of this space a test case covers, especially regions that are unique to distributed execution. As a result, testing repeatedly explores redundant scenarios, reducing overall effectiveness.

3 DistSQL Design

3.1 Architecture

To effectively detect logic bugs in distributed DBMSs, we address the above challenges and develop DistSQL, a differential testing framework tailored for this purpose, as shown in Figure 2. DistSQL first randomly initializes two database instances with the same data, one in centralized mode and one in distributed mode. It then continuously generates queries and executes them under both modes for differential testing. If feedback indicates no new distributed interaction after several consecutive executions, DistSQL performs database state mutation.

3.2 Testing Workflow

The testing workflow continuously generates and executes SQL statements. We implement an abstract syntax tree (AST) based SQL generator [1, 22, 45, 47] that constructs queries in a top down manner by first selecting the SQL statement type (e.g., `SELECT`, `CREATE TABLE`) and then recursively generating its components (e.g., `ORDER BY` clauses, expressions). Generation is guided by ***seeds***, where each seed is an integer sequence [22]. During generation, when there are n feasible choices, the next value s in the seed selects an option via $c = s \bmod n$. For example, when a function needs to be chosen from several options (e.g., `sum`, `avg`, `min`, `max`), we have $n = 4$. If $s = 9$, then $c = 1$, and thus the `avg` function (counting from 0) is selected. This enables reproducible and controllable mutation of test cases. Additionally, we reduce the likelihood of generating overly complex predicates and subqueries (e.g., by limiting the AST depth) to avoid empty results. DistSQL follows a five step testing loop, as detailed below.

(S1) Seed Mutation: Select and mutate a batch of seeds from the *Seed Pool* according to their assigned priority (§ 6).

(S2) SQL Generation and Execution: The *SQL Generator* produces a batch of read only queries, which the *Concurrent Executor* runs under centralized and distributed modes of the DBMS.

(S3) Bug Detection: The *Differential Checker* compares results from centralized and distributed executions and flags any *mismatch* as a potential logic bug.

(S4) Feedback and Seed Pool Update: (**S4.1**) The *Runtime Monitor* extracts query plan features (§ 5.1) as feedback. (**S4.2**) The *Feature Pool* is updated to track logic coverage and identify **interesting** test cases, namely those that exercise previously unseen features, which indicates new distributed interactions. (**S4.3**) Seeds for these interesting test cases are added to the *Seed Pool* for future mutation.

(S5) State Mutation and Iteration: If no new features are discovered after several iterations, the current database state is considered **uninteresting**. In this case, we modify the database state to promote exploration of new execution behaviors. This process uses data-modifying statements such as `UPDATE` and `DELETE`, as well as data definition statements (DDL) such as `CREATE` and `ALTER`. By comparing the contents and schemas of all tables across the two databases, we can potentially find logic bugs introduced by these statements. The process then returns to Step **(S1)**.

To address **C1**, DistSQL strategically performs database state mutation to elicit diverse distributed execution. In Step **(S2)** and Step **(S5)**, it controls data placement across nodes and generates SQL that references distributed data. This design triggers a broad set of distributed execution paths and increases the likelihood of exposing distributed logic bugs.

To address **C2**, DistSQL captures high-level execution semantics by analyzing query plans. In Step **(S4)**, it extracts query plan features from each plan to characterize the execution logic exercised by the query for feedback. This reduces redundancy and steers testing toward previously unexplored execution logic. Moreover, this feedback mechanism is black-box and requires no code instrumentation. Finally, in Step **(S1)** and Step **(S5)**, DistSQL adaptively prioritizes interesting seeds and modifies the database state, to progressively uncover more distributed behaviors, regardless of DBMS implementation details.

4 Distributed Database State Mutation

Distributed database state mutation serves two purposes: (1) to initialize the distributed database state at the start of testing; and (2) to mutate it when the feedback (§ 5) indicates that the current state is less effective. The mutation helps trigger more distributed behaviors.

We define the ***distributed database state*** as the combination of components that describe not only the data and its schema but also, crucially, how the data is distributed across nodes, including: (1) the table schema, such as column types and primary keys; (2) table content, i.e., the data stored in tables; (3) table configurations, such as partitioning, replication, and storage engines; and (4) auxiliary state, including statistics, indexes, and execution-related configurations such as the number of workers and optimization rules.

The state mutation is largely general, independent of different systems, as most operations (e.g., inserts and updates) share similar forms across systems, and schema operations like CREATE TABLE follow common structures for column definitions and primary keys. Only modest DBMS-specific adaptation is required (e.g., handling special table options such as partitioning or sharding).

Mutation principles. (1) To trigger more distributed behaviors, we aim for diverse and controllable data distribution. Automatic partitioning by the DBMS is often insufficient. For instance, if tables

have few rows, the system may place all data on a single node despite multiple nodes being available. (2) We also mutate configurations that affect distributed execution, such as table configurations and executor concurrency level, to further diversify execution paths.

Controlling data distribution. We explicitly control how data is distributed, aiming to separate rows across nodes to trigger distributed execution. While the actual placement remains randomized, it is indirectly guided by our feedback mechanism: uninteresting distributed database states are identified and mutated to encourage more meaningful execution paths.

The data distribution process involves three main steps: ❶ *creating partitioned tables*, ❷ *controlling partition placement*, and ❸ *inserting data into appropriate partitions*.

For step ❶, most DBMSs offer various partitioning options, such as hash, range, or manual partitioning, enabling explicit control over how tables are partitioned.

For step ❷, we support two strategies depending on the capabilities of the DBMS: (1) Assigning storage constraints to entire tables or specific partitions to pin them to designated nodes. For example, in CockroachDB, this can be achieved using `ALTER TABLE ... CONFIGURE ZONE`. (2) Creating local tables on individual nodes and defining a distributed view over them. For example, `Distributed` engine in ClickHouse implicitly creates such a view.

For step ❸, we insert data into randomly selected partitions. When the entire table is pinned (via storage constraints or local tables), data is inserted directly. Otherwise, for partitioned tables, we influence data placement by selecting appropriate partition key values; for example, in range-partitioned tables, keys are chosen to fall into specific ranges mapped to corresponding partitions. The inserted values consist of a mix of randomly generated data, subquery results, and carefully selected boundary values (e.g., `NULL`, `0`, and values near overflow thresholds), increasing the likelihood of triggering corner-case behaviors.

Other distributed database state mutation. In addition to controlling data distribution, we apply other types of state mutation to increase execution diversity:

- Modifying storage engines or table configurations that influence distributed execution, such as enabling clustered indexes in TiDB or configuring the number of tablets in YugabyteDB;
- Applying auxiliary operations such as creating indexes, updating statistics, or enabling optional components. For example, adding a TiFlash replica, where TiFlash is a columnar engine in TiDB for analytical workloads, may introduce additional data movement and increase cross-node execution; and
- Creating views or tables from query results, as well as performing row-level updates or deletions, to further enrich schema complexity and increase the diversity of data.

While we do not directly control operator-level properties such as join types or data shuffling mechanisms, these aspects are not ignored. By adjusting the aforementioned states and generating diverse queries, the DBMS optimizer naturally explores varied physical plans, such as different join types, shuffling mechanisms, and execution strategies (e.g., hash or sort aggregation).

5 Feedback for Execution Space Exploration

5.1 Overview

Feedback directs the search toward promising test cases, whereas unguided exploration quickly becomes redundant and inefficient. Our feedback relies on ***query plan features***. A *query plan feature* summarizes a ***query plan*** to capture characteristics of distributed execution, in particular logic unique to distributed systems. It focuses on operator interactions, namely how each operator obtains data from its children and propagates results to its ancestors, because operations may run on different nodes. A query plan is the execution flow of an SQL statement in centralized or

distributed mode. It is a tree whose nodes are basic *operators* (e.g., `Join`, `Sort`, `Filter`), with data flowing from the leaves to the root that returns the result.

Query plan features serve two main purposes: ① to identify *interesting* test cases that trigger unexplored logic specific to distributed execution; ② to recognize *uninteresting* database states (i.e., when no new features are discovered after several executions) and mutate them to provoke more diverse distributed behaviors.

Each feature is encoded as a ***<pattern, score>*** pair. A ***pattern*** (§5.3) denotes the operator subgraph that participates in the distributed interaction, and ***score*** (§5.4) quantifies its relevance to distributed execution. Thus, new patterns or score updates on old patterns in query plan features are all considered *interesting* for feedback.

5.2 Feedback Workflow

We now elaborate on the feedback-relevant parts of testing workflow (§ 3.2), including steps of (**S2**), (**S4**), (**S5**), and (**S1**). Together they form a feedback guided testing loop.

(S2) From seed to query plans: Each query is executed under both centralized and distributed modes, and the two corresponding ***query plans*** are collected. To encourage distributed behavior, we bias generation toward special query structures, such as multi-table joins, nested subqueries, window functions, and aggregations, which are more likely to trigger cross-node communication.

(S4.1) Feature extraction: From the query plans generated under the centralized and distributed modes, we independently extract structural features to produce two sets of ***query plan features***, each summarizing the execution logic of its respective plan (§ 5.5). These feature sets are then used to assess logic coverage and identify behavioral differences between the two execution modes.

(S4.2) Feature pool update: The extracted features from both modes are used to update the ***feature pool***, which tracks coverage of distributed execution logic. If any previously under-explored logic is exercised (§ 5.6), the pool is updated accordingly.

(S4.3) Keep interesting seeds: If the feature pool is updated, the test case and its associated seed are marked as *interesting* for having triggered new distributed behaviors. The seed is added to the ***seed pool*** for further mutation.

(S5) Database state mutation: If no interesting seeds are identified after several iterations, we trigger a database state mutation to escape from current state and promote the discovery of new execution paths. (§ 4).

(S1) Seed mutation: To continue testing, a seed is selected from the seed pool and randomly mutated to generate a new SQL query. This enables the exploration of meaningful variations of previously successful queries, progressively driving the system into various distributed states.

5.3 Plan Patterns based on Operator Interactions

A key problem here is to determine which subgraphs (patterns) of a query plan, or whether the entire query plan, should be used as the pattern in a query plan feature for feedback.

Entire query plan as pattern. One approach is to treat the entire query plan as a pattern [2]. However, it is obviously overly sensitive, particularly under diverse SQL inputs (§ 2.2), where nearly half of the queries result in unique plans. In practice, many query plans appear different but are actually repetitive in nature. They share similar core structure and semantics, with only minor variations (e.g., a few operators replaced or reordered). This motivates the need for a more fine-grained and semantics-aware representation of execution logic.

Basic operators as pattern. An alternative is to track which operators appear in the query plan. However, focusing on individual operators discards the interactions between operators and can quickly saturate, i.e., most operators can be covered with only a few test cases. Tracking adjacent

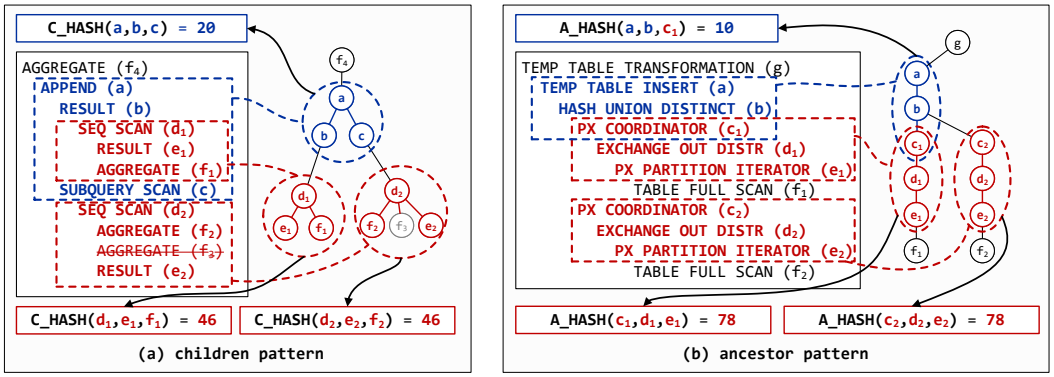


Fig. 3. Plan patterns for feedback. Each pattern is illustrated with a query plan on the left and its corresponding tree structure on the right. In (a), for example, nodes are labeled as a, d_1, and d_2, where identical letters indicate the same operator type: a denotes APPEND, and d_1, d_2 denote SEQ SCAN.

operator pairs can help a little but still misses important interaction semantics, such as those involving multiple children or indirect ancestors. For example, operators like HashJoin aggregate data from multiple children, and others (e.g., EXCHANGE OUT DISTR) exhibit meaningful behavior through grandparent-child interactions, both of which are critical for distributed execution.

Patterns used in DistSQL. Operator interactions, describing how an operator obtains data from its children and passes data to its parent, are often trivial in centralized execution but can differ significantly in distributed execution due to cross-node coordination and data shuffling. We aim to further explore interactions that are unique to distributed execution but are missed by the above approaches. Therefore, we design the *query plan feature* based on operator interactions, including two pattern types:

(1) *Children pattern.* This pattern considers an operator with all its children, capturing how the operator interacts with all its children, as the aggregation may involve distributed execution. Since an operator's behavior is generally unaffected by the order or duplication of its children, we deduplicate and sort child operators by their names, without considering detailed parameters, before forming the pattern with their parent. This level of abstraction is sufficient in practice; otherwise, the search space would grow excessively and the exploration could be trapped in overly fine-grained variations. Figure 3(a) presents three example patterns, and we use the C_HASH function to uniquely identify each pattern. For example, C_HASH(a, b, c)=20 and C_HASH(d_1, e_1, f_1)=46. The blue frame shows a unique children pattern; the red frame shows that repeating or reordering children like AGGREGATE and RESULT does not affect the pattern hash. Some patterns (e.g., (f_4, a) or (b, d_1)) are omitted for clarity.

(2) *Ancestor pattern.* This pattern includes an operator along with its parent and grandparent, highlighting indirect interactions that may reflect distributed semantics. As with the children pattern, we compute the hash using only operator names to maintain a consistent abstraction level. As shown in Figure 3(b), we present three example patterns and compute their hash values using the A_HASH function. For instance, A_HASH(a, b, c_1)=10 and A_HASH(c_1, d_1, e_1)=78. The blue frame shows a unique ancestor pattern, and the red frame shows that structurally similar interactions produce the same pattern hash.

Algorithm 1 Query Plan Feature Extraction

Input: Query plan tree T;
its operators $O = \{o_1, o_2, \ldots, o_n\}$
Output: Query plan features F

1: $F \leftarrow \emptyset$
2: **for all** operator $o_i \in O$ **do**
3: $s(o_i) \leftarrow \text{Score}(o_i)$
4: $n(o_i) \leftarrow \text{NumericID}(o_i)$
5: **end for**
6: **for all** operator $o_i \in O$ **do**
7: $C \leftarrow \text{Sort}(\text{Deduplicate}(\text{Children}(o_i)))$
8: $h_c \leftarrow \text{HashRange}(n(o_i), n(c_1), n(c_2), \ldots)$
9: $s_c \leftarrow \text{Avg}(s(o_i), s(c_1), s(c_2), \ldots)$
10: $F \leftarrow F \cup \{(h_c, s_c)\}$
11: **end for**
12: **for all** operator $o_i \in O$ **do**
13: $p \leftarrow \text{Parent}(o_i)$
14: $g \leftarrow \text{Parent}(p)$
15: $h_a \leftarrow (n(g) \ll 20) \oplus (n(p) \ll 10) \oplus n(o_i)$
16: $s_a \leftarrow \text{Avg}(s(g), s(p), s(o_i))$
17: $F \leftarrow F \cup \{(h_a, s_a)\}$
18: **end for**
19: **return** F

5.4 Distributed Execution Score

We assign a score to each pattern to reflect its relevance to distributed execution. The pattern score is the average across all scores of operators within the pattern. For each operator, the operator score is calculated as follows:

$$\textbf{\textit{score}} = \textbf{\textit{node}} \times \textbf{\textit{concurrency}} \times \textbf{\textit{volume\_coeff}}$$

Here, ***node*** denotes the number of nodes involved in executing the operator, ***concurrency*** indicates how many workers or threads are involved in executing this operator, and ***volume_coeff*** reflects the data scale processed by the operator.

The formula emphasizes distributed behavior: operators involving more nodes or higher concurrency receive higher weights, while those processing little or no data are downweighted using a piecewise-defined function *volume_coeff*. For each operator, we assign *volume_coeff* $= 0.1$ if its result is empty, 1 if it outputs fewer than 10 records, and 1.5 if it outputs more than 10 records.

These values are extracted from the query plan whenever available. As our approach treats the DBMS as a black box, some metrics may be unavailable and are then set to the default value of 1. For example, in CockroachDB, each operator is annotated with the set of nodes responsible for its execution. In TiDB, certain operators such as `IndexJoin` expose concurrency information, such as the number of workers. The number of rows processed by each operator is reported as, for example, `ActualRows` in the query plan.

5.5 Query Plan Feature Extract Algorithm

Algorithm 1 outlines the extraction of both pattern types from the query plan tree, along with the computation of their scores. As a preliminary step, each operator o_i is assigned a distributed execution score and a unique ID $n(o_i)$, computed by hashing its name with C++'s `std::hash` (Lines 2–5). For operators involving inter-node data shuffling, we append the shuffle method (e.g.,

`broadcast`) to the operator name to more clearly distinguish distributed logic. As a limitation, this information is unavailable if the DBMS does not expose it, which may lead to different shuffling logic being mapped to the same pattern, introducing some ambiguity in scoring.

To compute hash for *children pattern* (`C_HASH`), the algorithm retrieves the children C of each operator o_i, removes duplicates, and sorts them into a canonical order. The numeric IDs of both the operator and its processed children are then combined using `HashRange` (a sequence hash implemented using `boost::hash_range`) to compute the hash. The score of this pattern is obtained by averaging the scores of the operator and its children (Lines 6–11).

To compute hash for *ancestor pattern* (`A_HASH`), the algorithm retrieves the parent p and grandparent g of operator o_i. Their numeric IDs $n(g)$, $n(p)$, and $n(o_i)$ are shifted left by different offsets and then XORed together to produce the ancestor hash (using a default ID value (e.g., 0) if either is missing). The score for this pattern is similarly calculated as the average of individual scores of its operators (Lines 12–18).

5.6 Feature Pool

After the feature extraction process, we obtain two sets of features from the query plans produced by the centralized and distributed modes of the DBMS, denoted as $\boldsymbol{F_c}$ and $\boldsymbol{F_d}$, respectively. These feature sets are used for two purposes: (1) to identify whether the SQL query triggers new patterns or patterns with higher scores than previously recorded, indicating newly covered execution logic or logic more closely related to distributed execution; and (2) to detect new ***feature differences*** between the two modes by computing the *symmetric difference* of $\boldsymbol{F_c}$ and $\boldsymbol{F_d}$ (ignoring scores during this computation). By comparing features across centralized and distributed modes, we can identify discrepancies in execution logic between the two modes, allowing us to focus on previously under-explored logic differences. These differences may further reveal new result variations that are valuable for differential testing.

To support these objectives, we maintain two separate pools: $\boldsymbol{P_{all}}$ for tracking all *features* covered by a SQL query, and $\boldsymbol{P_{diff}}$ for tracking *feature differences*. These pools share similar internal logic and store features by their pattern hashes, along with the historical maximum *distributed execution scores*, which are initialized to zero. A pool is updated when any feature it tracks is either previously unseen or appears with a higher score than previously recorded. In such cases, the historical score is updated and the corresponding SQL query is marked as *interesting*.

If a SQL query is marked as *interesting*, the *seed* used to generate it is likewise considered *interesting* and is added to the *Seed Pool* for future mutation and exploration. If no interesting seeds are identified after several iterations, we trigger a database state mutation to promote exploration of new execution behaviors.

6 Implementation

We implement DistSQL in nearly 28,000 lines of C++ code. DistSQL supports six distributed DBMSs: OceanBase, TiDB, CockroachDB, YugabyteDB, ClickHouse, and a commercial one. Adding a new system typically involves configuring built-in functions, disabling unsupported syntax, and adding dialect rules. For differential testing, we first compute a hash for each result row and compare the sets of row hashes for efficiency. If a mismatch is found, the results are re-compared after sorting to avoid false positives.

Result Stability. Distributed DBMSs may yield varying outputs permitted by flexible SQL semantics, primarily due to non-deterministic execution and floating-point instability. To ensure reliable testing, we enforce two rules: (1) **Ordering enforcement:** We enforce result ordering across all columns when required, such as for queries with `LIMIT` or window functions like `RANK()`. We also

Table 3. Tested DBMSs.

System	Version	Languages	SQL Compatibility
OceanBase [37]	4.3.0.1	C++	MySQL
TiDB [41]	7.5.1	Go, Rust	MySQL
CockroachDB [11]	23.1.19	Go	Postgres
YugabyteDB [59]	2.21.1	Java, C++	Postgres
ClickHouse [9]	24.3.3.102	C++	ClickHouse
Commercial	/	C++	MySQL/Postgres

avoid multiple window functions in one `SELECT`. (2) **Floating-point restrictions:** We avoid using floats in equality checks, `IN`, `GROUP BY`, or `DISTINCT`/`UNION`, and avoid `LIMIT` over ordered float columns, as small differences can cause divergent outputs. It reduces diversity to some extent but effectively eliminates a large number of false positives that would otherwise occur.

Seed pool. We implement the seed pool as a priority queue. We assign higher priority to seeds that exercise more previously unseen logic. These are mutated first to generate new test cases, improving bug-discovery efficiency. Each seed's priority is its initial score, based on how many new features it triggers, multiplied by its *interesting mutation rate*, defined as the proportion of its mutations marked as interesting.

Seed mutation. The seed sequence is consumed sequentially during SQL generation. Mutations at positions that affect high-level structural choices (e.g., selecting the query type) can change how subsequent seed elements are interpreted, potentially altering large parts of the generated SQL structure even if the remaining seed elements themselves are unchanged. We therefore assign lower mutation rates to these positions. Occasional large-scale mutations are still applied to expand the search space.

7 Evaluation

7.1 Experimental Setup

Tested DBMSs. We evaluate DistSQL on five widely used open-source distributed DBMSs and one commercial system. Table 3 provides an overview of these systems, which are implemented in different programming languages and widely adopted in industry.

Centralized vs. distributed testing. We apply differential testing by deploying the same DBMS in two modes: a centralized mode and a distributed mode. In the *centralized* mode, the system is deployed in minimal form, typically with a single process running on one node. If a coordinator is required, it is co-located on the same node. In the *distributed* mode, DBMSs often comprise multiple components with distinct responsibilities. We assign three nodes to the core components (such as data storage and parallel execution), and allocate an additional node to each remaining component (such as coordinators or auxiliary workers). The use of three nodes captures a critical transition from centralized to distributed execution, where the differences in behavior become significant. Beyond three nodes, the marginal increase in execution diversity tends to be smaller [57]. To simulate a distributed environment, each node is deployed in a separate Docker container. All containers are connected via a virtual subnet and run on the same physical machine.

Comparison. We compare DistSQL against SOTA database and distributed system testing tools, including Squirrel [65], QPG [2], and EET [24], as well as distributed system testing tools, including Jepsen [20] and Mallory [36]. (§ 7.4)

Machine configuration. All experiments are conducted on a machine with an AMD EPYC 7J13 64-Core Processor (2.45 GHz), 512 GB of RAM, running Ubuntu 22.04. All systems are tested with their latest versions at evaluation time.

7.2 Open-Source Distributed DBMS Testing

Each DBMS is tested three times, with each test lasting one week. We manually analyze root causes using query plans and developer feedback. After implementing *result stability* (§ 6), **no false positives** are reported by DistSQL.

Table 4. Types of the found logic bugs.

System	Distributed	Both	Fixed	Confirm	All
OceanBase	4	10	14	14	14
TiDB	9	8	8	16	17
CockroachDB	1	1	1	2	2
YugabyteDB	0	2	1	1	2
ClickHouse	6	4	6	8	10
Total	20	25	30	41	45

Bug types. Table 4 summarizes all 45 logic bugs found by DistSQL. Of these, 41 have been confirmed and 30 have already been fixed. For the 16 bugs we could track, all were fixed with 14 patches. All bugs were identified through mismatches between centralized and distributed results. Among them, 20 appear only in distributed mode, while the other 25 can occur in both modes. At the time of discovery, the two modes either produced different incorrect outputs, or one occasionally returned a correct result when a bug-free execution path was taken. We manually checked whether their semantics were consistent with the results in each mode to determine whether a bug existed. For those that can occur in both modes, the distributed mode typically triggers them more frequently due to higher concurrency levels, more query plan variations, and more different execution paths.

Table 5. Root causes of logic bugs in the distributed mode.

System	ID	DistOpt(7)	DistImpl(8)	Concurrency(6)	Float(7)
OceanBase	1-3	●	○	○	○
	4	○	○	●	○
TiDB	5	●	○	●	○
	6-8	○	●	○	○
	9-12	○	●	○	●
	13	○	○	●	●
CockroachDB	14	●	○	○	○
ClickHouse	15-16	●	○	○	○
	17	○	●	○	●
	18-19	○	○	●	○
	20	○	○	●	●

Bug root causes. We manually analyze 45 logic bugs and categorize their root causes. Table 5 lists those specific to distributed mode. ❶ *DistOpt*: 7 bugs are related to distributed query optimization. These arise when the optimizer generates incorrect plans due to the insufficient handling of corner cases in distributed execution. For example, subqueries in OceanBase, joins in ClickHouse, and `UNION ALL` operations in CockroachDB trigger incorrect distributed behavior, resulting in wrong query results. ❷ *DistImpl*: 8 bugs stem from faulty implementations of distributed components. These modules are less frequently exercised, leaving corner cases poorly addressed. They often re-implement functionality already available in other components to support distributed processing but do so inconsistently. For instance, some expressions yield different results when used as columns versus predicates. ❸ *Concurrency*: 6 bugs are caused by issues in concurrent execution. These

Table 6. Results of sensitivity experiments.

System	DistSQL *Random*				DistSQL *Text*				DistSQL *JoinAware*				DistSQL			
	Text	*Feature*	*LBug*	*DLBug*	*Text*	*Feature*	*LBug*	*DLBug*	*Text*	*Feature*	*LBug*	*DLBug*	*Text*	*Feature*	*LBug*	*DLBug*
OceanBase	2.0M	20,171	7	2	1.8M	21,020	8	3	2.0M	33,407	10	3	2.0M	36,935	10	3
TiDB	360K	5,383	4	3	1.1M	8,044	7	5	1.3M	12,768	11	7	1.4M	11,495	11	7
CockroachDB	814K	14,439	0	0	1.8M	13,552	0	0	3.9M	19,263	1	1	3.7M	16,925	1	1
YugabyteDB	1.3M	10,928	2	0	1.6M	12,302	2	0	2.7M	12,824	2	0	2.8M	17,472	2	0
ClickHouse	2.0M	657	5	2	1.5M	651	6	3	1.6M	664	9	6	2.0M	659	9	6
Total	6.5M	51,578	18	7	8.0M	55,569	23	11	11.5M	78,926	33	17	11.9M	83,486	33	17

Table 7. Results of comparison experiments.

Program	Jepsen			Mallory			QPG				EET				DistSQL			
	Text	*Feat.*	*LBug*	*Text*	*Feat.*	*LBug*	*Text*	*Feat.*	*LBug*	*DLBug*	*Text*	*Feat.*	*LBug*	*DLBug*	*Text*	*Feat.*	*LBug*	*DLBug*
OceanBase	3	9	0	3	9	0	14,944	560	0	0	7,781	8,559	1	0	2,070,192	36,935	10	3
TiDB	2	4	0	2	4	0	31,832	2,467	4	1	4,148	4,807	1	0	1,374,844	11,495	11	7
CockroachDB	2	4	0	2	4	0	3,348	996	0	0	3,240	6,251	0	0	3,657,007	16,925	1	1
YugabyteDB	3	4	0	3	4	0	17,014	477	1	0	872	3,458	0	0	2,769,943	17,472	2	0
ClickHouse	2	2	0	2	2	0	1,650	79	0	0	43,941	648	0	0	1,994,544	659	9	6
Total	12	23	0	12	23	0	68,788	4,579	5	1	59,982	23,723	2	0	11,866,530	83,486	33	17

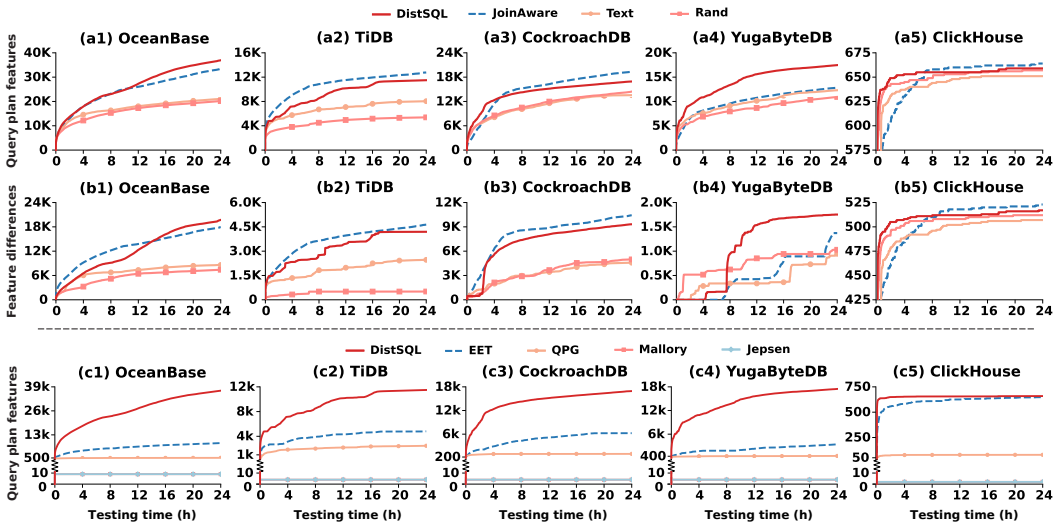


Fig. 4. Coverage growth. (a)–(b): Sensitivity experiments; (c): Comparison with existing approaches.

bugs are rarely triggered in centralized mode with relatively lower concurrency but occur more frequently in distributed mode with higher concurrency. ❹ *Float*: 7 of these bugs involve floating-point operations, showing that our approach effectively handles floating-point non-determinism, enabling diverse test cases without false positives.

7.3 Sensitivity Analysis

To evaluate the effectiveness of our feedback mechanism based on query plan feature, we implement three variants of DistSQL: (1) *DistSQL* $_{Random}$, which generates random SQL queries without feedback, (2) *DistSQL* $_{Text}$, which uses the query plan text-guided feedback identical to that of QPG [2], with its paper and code referenced, and (3) *DistSQL* $_{JoinAware}$, identical to DistSQL except that it does not sort the children of join operators, as their order may affect the underlying logic.

We run a 24-hour experiment with these tools and DistSQL. Table 6 summarizes the number of query plan texts, features (Feat.) extracted by DistSQL, and two bug metrics: LBug (total logic bugs) and DLBug (logic bugs unique to the distributed mode).

Testing coverage. DistSQL achieves substantially broader testing coverage than *DistSQL* $_{Random}$ and *DistSQL* $_{Text}$, discovering 83% and 49% more query plan texts, and extracting 61% and 50% more query plan features, respectively. Query plan text-guided feedback can mark up to 50% of test cases as interesting, often resulting in redundancy. In contrast, DistSQL's query plan feature-guided feedback filters out over 99% of redundant cases, concentrating on unexplored distributed logic. This enables a compact and efficient seed pool of 1,000 to 3,000 seeds, improving overall testing efficiency. As shown in Figure 4(a) and (b), DistSQL consistently achieves higher feature coverage and discovers more feature differences than *DistSQL* $_{Random}$ and *DistSQL* $_{Text}$. The variant *DistSQL* $_{JoinAware}$ shows no consistent improvement in coverage across DBMSs, sometimes higher and sometimes lower than DistSQL.

Bug Detection. During the 24-hour testing period, DistSQL detects 33 logic bugs, including 17 DLBugs that occur only in the distributed mode. In comparison, *DistSQL* $_{Random}$ and *DistSQL* $_{Text}$ detect 15 and 10 fewer logic bugs, and 10 and 6 fewer DLBugs, respectively. These results demonstrate the effectiveness of query plan feature for capturing execution behavior, reducing redundancy, and enhancing both bug detection and testing coverage. The additional variant *DistSQL* $_{JoinAware}$ reports the same set of bugs as DistSQL. Although it may explore more join combinations, this does not necessarily reveal new bugs. Its uneven coverage across DBMSs indicates that it may sometimes overemphasize fine-grained differences and slightly reduce coverage in other aspects.

7.4 Comparison

We evaluate five open-source distributed DBMSs (Table 3) using four existing tools: Jepsen [20], Mallory [36], QPG [2], and EET [24], and compare their performance with DistSQL.

These tools are selected from existing open-source approaches to ensure a representative and meaningful comparison. Among distributed system testing approaches, Jepsen and Mallory are the most closely related to DBMS testing. Among recent DBMS testing tools, QPG employs feedback mechanism related to query plan, while EET achieves higher SQL diversity than many prior approaches [24]. Since none of the tools support all DBMSs, we extend them for compatibility. Each system is tested for 24 hours, with results summarized in Table 7.

Testing coverage. Our system achieves significantly higher coverage, with query plan text coverage 197 times that of EET and query plan feature coverage 3.5 times that of EET. It also surpasses Jepsen, Mallory, and QPG by orders of magnitude in both metrics. These results underscore DistSQL's effectiveness in thoroughly exploring a wide range of execution paths within the vast execution space of distributed DBMSs.

DistSQL's higher testing coverage comes from two primary factors. First, its *distributed diversity oriented database state mutation* actively mutates database states to trigger more execution paths. Second, its *distributed interaction guided execution space exploration* mechanism effectively identifies unique distributed logic, avoiding redundant test cases and enabling efficient exploration of the explosive execution space.

Figure 4(c) illustrates the rapid growth of covered query plan features in early testing phases, stabilizing later. Across all stages, DistSQL maintains superior feature coverage.

Bug detection. Jepsen and Mallory fail to find any new logic bugs. DistSQL identifies 28 more bugs than QPG and 31 more than EET. Notably, 16 of these bugs are specific to distributed logic and do not manifest in centralized mode, highlighting the importance of targeting distributed execution paths. While Jepsen and Mallory are effective at finding certain types of bugs, many of the bugs they found have already been fixed in current versions. In contrast, DistSQL is able to find additional logic bugs beyond those detected by existing approaches.

7.5 Case Studies of the Found Bugs

This section highlights three logic bugs manifested only in distributed mode.

Fig. 5. Logic bug example 1.

```
--- Statements for database initialization
CREATE TABLE t1 as (SELECT null::int4 as c1, null::int4 as c2 WHERE false);
ALTER TABLE t1 CONFIGURE ZONE USING constraints = '[+region=r1]';
INSERT INTO t1 (c1, c2) VALUES (1, 1);
CREATE INDEX idx1 on t1 (c2);

CREATE TABLE t2 (c1 int4 PRIMARY KEY, c2 bool);
ALTER TABLE t2 CONFIGURE ZONE USING constraints = '[+region=r3]';
INSERT INTO t2 (c1, c2) VALUES (1, null), (2, null), (3, null), (4, null);

--- Bug-triggering statement
SELECT false as c0 FROM t1
  WHERE (SELECT 1) is NULL
UNION ALL
  (SELECT DISTINCT EXISTS (SELECT 1 WHERE false)
  FROM t2)
ORDER BY c0 LIMIT 2;
--- Unexpected (produced by distributed systems)
false, false
--- Expected (produced by single node)
false
```

Incorrect distributed logic in CockroachDB. Figure 5 illustrates a logic bug in the CockroachDB. The first query selects a constant value, `false`, from table `t1`, with a predicate that is always false, resulting in an empty set. The second query includes a subquery that also evaluates to an empty set. Consequently, the `EXISTS` operator should yield `false`. As the `DISTINCT` quantifier is applied, the query should return a single row with the value `false`.

Unexpectedly, CockroachDB (with three nodes) outputs two rows with the value `false`. Increasing the `LIMIT` parameter further amplifies this discrepancy, with the output reaching up to four rows (the total in `t2`). This behavior is inconsistent with the expected result. The bug reliably manifests under specific conditions: when the optimizer selects `t2` to be read from a remote node, `t1` from another node, and performs the `UNION ALL` operation on a node other than the one hosting `t2`. Consequently, this bug cannot be triggered in the centralized mode, as the bug arises from the interaction between nodes in a distributed environment.

Fig. 6. Logic bug example 2.

```
--- Statements for database initialization
CREATE TABLE t1 (c1 INT, c2 INT);
ALTER TABLE t1 SET tiflash replica 1;
INSERT INTO t1 (c1, c2) VALUES (-82, -4), (1, 4), (-11, -45), (-20, 142);

--- Bug-triggering statement
SELECT
  c2,
  lag(1) over (PARTITION BY c1 ORDER BY c2) as a,
  substring('abc', c2) as b,
  hex(substring('abc', c2)) as bh
FROM t1;
--- Unexpected (produced by distributed component)
{4 | NULL | '\0abc' | 00616263 }...
--- Expected (produced by local environment)
{4 | NULL | '' | '' }...
```

Incorrect distributed component implementation in TiDB. Figure 6 highlights a logic bug in TiDB. The query selects four columns, with the second and third columns being critical to

the bug. The first column is included to display the value of c2, helping to clearly illustrate the bug. The fourth column contains the hexadecimal representation of the third column's substring value, offering a precise view of the underlying string data. The second column plays a role in triggering the bug, while the third column demonstrates the unexpected behavior. According to TiDB's documentation, which claims MySQL compatibility, the substring function should return an empty string if the second parameter exceeds the string's length. However, in a distributed environment, the function instead returns a special string with a leading \0.

Moreover, with a carefully crafted query, the substring function can read beyond the string's boundary, even accessing data from other rows. This behavior not only deviates from expected functionality but also raises potential security concerns, as it may expose sensitive data.

Fig. 7. Logic bug example 3.

```
--- Statements for database initialization
CREATE TABLE t0 on cluster default (c1 Int32 PRIMARY KEY, c2 Bool);
CREATE TABLE t1 on cluster default as t0 ENGINE = Distributed(default, ch_main, t0, c1);

ALTER TABLE t0 on cluster default add column c3 Float64;
ALTER TABLE t1 on cluster default add column c3 Float64;

INSERT INTO t1 (c1,c2,c3) VALUES
  (-385119672 , false, 44.24),
  (-2070018982, false, -0.3 ),
  (2112636050 , true , 79.89);
--- (omit 10 insertions to t1)

--- Bug-triggering statement
SELECT DISTINCT first_value(c1) over (PARTITION BY c2 ORDER BY c1)
FROM t1;
--- Unexpected (produced by distributed table)
-385119672, -2070018982, -2112636050
--- Expected (produced by simple table)
-2070018982, -2112636050
```

Incorrect concurrent processing in ClickHouse. Figure 7 presents a logic bug in ClickHouse related to concurrent processing. The example begins by creating a distributed table on a cluster, configured to read data from local tables on individual nodes. A query is then executed to select the first value of c1 for each partition of c2, ordered by c1. Given that c2 has only two distinct values, true and false, the query should produce two partitions. Applying the DISTINCT quantifier to these partitions should result in two rows, each containing the first value of c1 for one partition.

However, the output contains three rows instead of two, deviating from the expected result. The discrepancy stems from incorrect handling of hash updates during parallel execution of the window function. While the function behaves correctly on a local table with limited concurrency, the distributed context introduces boundary cases that are not properly managed, leading to the observable inconsistency. This bug highlights the value of differential testing in finding bugs caused by parallelism in distributed components.

7.6 Production Study

We conducted a month-long evaluation of a commercial distributed DBMS that processes nearly a billion transactions per minute and serves thousands of enterprises, including major banks. DistSQL runs in a **black-box** manner, without access to source code or internal details. DistSQL found 20 logic bugs: 18 appeared only in distributed mode, and 2 in both centralized and distributed modes. All were confirmed and fixed by the vendor. This shows that DistSQL is effective and applicable in real-world production settings.

7.7 Other System Bugs

By monitoring connection status, DistSQL can detect server crashes and find system-level bugs. In the five open-source distributed DBMSs mentioned earlier (§ 7.2), DistSQL found 52 such bugs, including 32 memory bugs (e.g., null pointer dereferences) and 20 assertion failures. Of these, 28 were confirmed and 16 have been fixed by developers. In a 24-hour comparison with SQLSmith [1] and Squirrel [65], DistSQL found 17 memory bugs and 10 assertion failures, while both baselines found none.

8 Discussions

DML testing. We did not find any bugs triggered purely by DML statements. There are several reasons for this. We did observe some inconsistencies caused by DML, but these queries also contained SELECT clauses, which alone were sufficient to trigger the issues; therefore, they were not classified as DML-related bugs. Apart from the above cases, our generated DML statements (those without SELECT clauses) had relatively simple structures and were executed less frequently than read-only queries, which may have limited the exploration of this space.

Limitations. ① It is hard for DistSQL to improve its coverage if the distributed DBMSs have complex configurations. Many features are difficult to trigger without knowing the system internals. It needs system-specific test inputs and configuration fuzzing [27, 64] to uncover configuration-sensitive behaviors. ② DistSQL is less effective at finding bugs in rare or error-prone scenarios, such as transient network failures or I/O exceptions, which often involve special error-handling logic. Software fault injection [4, 12, 21] can help by introducing controlled failures to improve coverage of these paths. ③ DistSQL struggles to detect bugs from accumulative errors that depend on long execution histories or specific operation sequences. This can be addressed by using checkpointing [6, 7], which captures intermediate states and helps analyze delayed or state-dependent failures.

9 Related Work

9.1 Coverage-Guided Fuzzing

General-purpose fuzzers have been widely used and improved along various dimensions, such as seed selection [14, 19, 39, 42] and structural mutation [5, 38, 40, 51]. For instance, CollAFL [14] improves path exploration by leveraging accurate coverage information and introducing seed selection strategies guided by memory accesses and unexplored branches. Zest [38] enhances input generation by transforming random-input generators into deterministic parametric generators, effectively mapping low-level mutations to high-level structural changes.

However, these approaches struggle to cope with the complex semantics and execution behaviors of distributed DBMSs, and their feedback is often insufficient.

9.2 DBMS Fuzzing

Fuzzing has shown strong potential in finding various types of bugs in DBMSs. Several approaches [1, 13, 22, 28, 65] focus primarily on memory bugs and assertion failures. Other approaches [2, 18, 23, 24, 30, 43–45, 48, 53, 61] aim to find logic bugs by validating the query processing logic in local DBMSs through test oracles, which typically compare the execution results of two logically equivalent or related SQL queries. In addition, SemConT [31] introduces a formal model of the database to provide a ground truth for validation. Other approaches [3, 29, 49] adopt differential testing strategies, executing the same SQL statements under different runtime configurations or query hints to identify inconsistencies.

Most of these approaches rely heavily on random testing, providing limited guidance during SQL generation and database state mutation. Some approaches incorporate testing-scenario-specific

feedback signals, such as formal rule coverage in SemConT [31] and graph similarity of test cases in TQS [53], while others adopt more generic runtime feedback, such as branch coverage [30] or query plan text [2]. However, these feedback signals are often insufficient for thoroughly exercising the complex behaviors of distributed DBMSs.

Consequently, existing approaches fall short when applied to distributed DBMSs, as they largely overlook the unique characteristics of distributed execution. To address these challenges, DistSQL is specifically designed to more effectively explore distributed behaviors and find logic bugs that manifest in distributed execution scenarios.

9.3 Distributed System Fuzzing

Many approaches have been proposed to test generic distributed systems. Some adopt formal verification techniques to prove system correctness with respect to specified models or invariants [46, 54, 56, 62], while others focus on identifying performance bottlenecks and bugs in fault recovery mechanisms [8, 17, 55, 63]. Additional approaches aim to find concurrency bugs and state inconsistencies by systematically exploring interleavings and injecting failures in concurrent or fault-prone scenarios [16, 20, 25, 26, 34, 36, 58]. More recently, a few works [33, 52] have begun investigating semantic bugs by checking for violations of high-level correctness properties under realistic distributed operations such as reconfiguration and failover. However, these semantic bugs remain distinct from the logic bugs that occur in the query processing pipelines of distributed DBMSs, which involve richer SQL semantics. Testing distributed DBMSs poses additional challenges due to their complex semantics. Effective testing requires generating diverse and comprehensive test cases. Existing methods often rely on fixed test input, leading to limited coverage of SQL semantics. Furthermore, while these approaches can detect many logic bugs, verifying the correctness of complex SQL execution results remains a significant obstacle, leaving many SQL-semantics-related logic bugs in distributed DBMSs undetected.

10 Conclusion

This paper presents DistSQL, a differential testing framework for effectively detecting distributed logic bugs in distributed DBMSs. DistSQL introduces two key techniques: *distributed diversity oriented database state mutation*, which covers diverse distributed execution paths, and a *distributed interaction guided execution space exploration* mechanism, which captures high-level execution semantics to guide exploration within the vast execution space. We evaluate DistSQL on five open-source distributed DBMSs and a commercial system, discovering 65 previously unknown logic bugs, 61 of which have been confirmed by developers. Notably, 38 of these manifest only in distributed mode. Compared to four state-of-the-art testing approaches, DistSQL demonstrates significantly higher effectiveness in both bug detection and coverage, highlighting its capability to find distributed logic bugs that existing tools overlook.

11 Acknowledgement

We thank our anonymous reviewers for their helpful and constructive feedback on earlier versions of the paper. This work was supported by Smart Grid-National Science and Technology Major Project with No.2025ZD0808500 and National Natural Science Foundation of China under Grant No.62572021.

References

[1] anse1. 2015. sqlsmith: A random SQL query generator. https://github.com/anse1/sqlsmith.

[2] Jinsheng Ba and Manuel Rigger. 2023. Testing database engines via query plan guidance. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2060–2071.

[3] Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.
[4] Jia-Ju Bai, Yu-Ping Wang, Jie Yin, and Shi-Min Hu. 2016. Testing error handling code in device drivers using characteristic fault injection. In *Proceedings of the 2016 USENIX Annual Technical Conference*. 635–647.
[5] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: synthesizing structure while fuzzing. In *Proceedings of the 28th USENIX Security Symposium*, Vol. 19.
[6] Mohamed-Slim Bouguerra, Thierry Gautier, Denis Trystram, and Jean-Marc Vincent. 2010. A flexible checkpoint/restart model in distributed systems. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics (PPAM)*. 206–215.
[7] Anton Burtsev, Prashanth Radhakrishnan, Mike Hibler, and Jay Lepreau. 2009. Transparent checkpoints of closed distributed systems in emulab. In *Proceedings of the 4th European Conference on Computer Systems (EuroSys)*. 173–186.
[8] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Ming Gu, Qing Liao, and Yu Jiang. 2024. Chronos: Finding Timeout Bugs in Practical Distributed Systems by Deep-Priority Fuzzing with Transient Delay. In *2024 IEEE Symposium on Security and Privacy (SP)*. 1939–1955. https://doi.org/10.1109/SP54263.2024.00109
[9] ClickHouse. 2025. ClickHouse® is a real-time analytics DBMS. https://github.com/ClickHouse/ClickHouse.
[10] clickhouse-use 2025. ClickHouse use cases. https://clickhouse.com/use-cases.
[11] cockroachdb. 2025. CockroachDB — the cloud native, distributed SQL database designed for high availability, effortless scale, and control over data placement. https://github.com/cockroachdb/cockroach.
[12] Kai Cong, Li Lei, Zhenkun Yang, and Fei Xie. 2015. Automatic fault injection for driver robustness testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*. 361–372.
[13] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-free DBMS fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[14] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: path sensitive fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*. 679–696.
[15] Yu Gao, Wensheng Dou, Dong Wang, Wenhan Feng, Jun Wei, Hua Zhong, and Tao Huang. 2023. Coverage guided fault injection for cloud systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2211–2223.
[16] Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. 2023. Acto: Automatic end-to-end testing for operation correctness of cloud system management. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 96–112.
[17] Diwaker Gupta, Kashi Venkatesh Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex Snoeren, and Geoffrey M Voelker. 2011. DieCast: Testing distributed systems with an accurate scale model. *ACM Transactions on Computer Systems (TOCS)* 29, 2 (2011), 1–48.
[18] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 345–358.
[19] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. Seed selection for successful fuzzing. In *Proceedings of the 2021 International Symposium on Software Testing and Analysis (ISSTA)*. 230–243.
[20] jepsen io. 2018. Jepsen: A framework for distributed systems verification, with fault injection. https://github.com/jepsen-io/jepsen.
[21] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2020. Fuzzing error handling code using context-sensitive software fault injection. In *Proceedings of the 29th USENIX Security Symposium*. 2595–2612.
[22] 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)*. 4949–4965.
[23] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting transactional bugs in database engines via {graph-based} oracle construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 397–417.
[24] Zu-Ming Jiang and Zhendong Su. 2024. Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 821–835.
[25] Ahmed Khoumsi. 2002. A temporal approach for testing distributed systems. *IEEE Transactions on Software Engineering* 28, 11 (2002), 1085–1103.
[26] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. 2014. {SAMC}:{Semantic-Aware} model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 399–414.
[27] Junqiang Li, Senyi Li, Keyao Li, Falin Luo, Hongfang Yu, Shanshan Li, and Xiang Li. 2024. ECFuzz: Effective Configuration Fuzzing for Large-Scale Systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software*

Engineering. 1–12.

[28] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-oriented DBMS fuzzing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 668–681.

[29] Jie Liang, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Chengnian Sun, and Yu Jiang. 2024. Mozi: Discovering DBMS bugs via configuration-based equivalent transformation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

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

[31] Shuang Liu, Chenglin Tian, Jun Sun, Ruifeng Wang, Wei Lu, Yongxin Zhao, Yinxing Xue, Junjie Wang, and Xiaoyong Du. 2024. Semantic Conformance Testing of Relational DBMS. *Proceedings of the VLDB Endowment* 18, 3 (2024), 850–862.

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

[33] Chang Lou, Yuzhuo Jing, and Peng Huang. 2022. Demystifying and checking silent semantic violations in large distributed systems. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 91–107.

[34] Jeffrey F Lukman, Huan Ke, Cesar A Stuardo, Riza O Suminto, Daniar H Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, et al. 2019. Flymc: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[35] Tao Lyu, Liyi Zhang, Zhiyao Feng, Yueyang Pan, Yujie Ren, Meng Xu, Mathias Payer, and Sanidhya Kashyap. 2024. Monarch: A Fuzzing Framework for Distributed File Systems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 529–543.

[36] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. 2023. Greybox Fuzzing of Distributed Systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1615–1629.

[37] oceanbase. 2025. OceanBase is an enterprise distributed relational database with high availability, high performance, horizontal scalability, and compatibility with SQL standards. https://github.com/Oceanbase/oceanbase.

[38] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with Zest. In *Proceedings of the 2019 International Symposium on Software Testing and Analysis (ISSTA)*. 329–340.

[39] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: optimizing OS fuzzer seed selection with trace distillation. In *Proceedings of the 27th USENIX Security Symposium*. 729–743.

[40] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering (TSE)* 47, 9 (2019), 1980–1997.

[41] pingcap. 2025. TiDB - the open-source, cloud-native, distributed SQL database designed for modern applications. https://github.com/pingcap/tidb.

[42] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *Proceedings of the 23rd USENIX Security Symposium*. 861–875.

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

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

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

[46] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: A separation-logic library for verifying distributed systems. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 113–129.

[47] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. Citeseer, 618–622.

[48] 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)*. IEEE, 2072–2084.

[49] Jiansen Song, Wensheng Dou, Yu Gao, Ziyu Cui, Yingying Zheng, Dong Wang, Wei Wang, Jun Wei, and Tao Huang. 2024. Detecting Metadata-Related Logic Bugs in Database Systems via Raw Database Construction. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1884–1897.

[50] sqlancer. 2020. Automated testing to find logic and performance bugs in database systems. https://github.com/sqlancer/sqlancer.

[51] Prashast Srivastava and Mathias Payer. 2021. Gramatron: effective grammar-aware fuzzing. In *Proceedings of the 2021 International Symposium on Software Testing and Analysis (ISSTA)*. 244–256.

[52] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic reliability testing for cluster management controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 143–159.
[53] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting logic bugs of join optimizations in dbms. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[54] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. 2023. Model checking guided testing for distributed systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 127–143.
[55] Haoze Wu, Jia Pan, and Peng Huang. 2024. Efficient exposure of partial failure bugs in distributed systems with inferred abstract states. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1267–1283.
[56] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. {DistAI}:{Data-Driven} automated invariant learning for distributed protocols. In *15th USENIX symposium on operating systems design and implementation (OSDI 21)*. 405–421.
[57] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. 2014. Simple testing can prevent most critical failures: An analysis of production failures in distributed {Data-Intensive} systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 249–265.
[58] Xinhao Yuan and Junfeng Yang. 2020. Effective concurrency testing for distributed systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1141–1156.
[59] yugabyte. 2025. YugabyteDB - the cloud native distributed SQL database for mission-critical applications. https://github.com/yugabyte/yugabyte-db.
[60] yugabyte-use 2025. What YugabyteDB Users Have to Say. https://www.yugabyte.com/about/.
[61] Chi Zhang and Manuel Rigger. 2025. Constant Optimization Driven Database System Testing. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–24.
[62] Tony Nuda Zhang, Travis Hance, Manos Kapritsos, Tej Chajed, and Bryan Parno. 2024. Inductive invariants that spark joy: using invariant taxonomies to streamline distributed protocol proofs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 837–853.
[63] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. 2021. Understanding and detecting software upgrade failures in distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 116–131.
[64] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. 2023. Fuzzing configurations of program options. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–21.
[65] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970.

Received July 2025; revised October 2025; accepted November 2025